

APPLICATION
FOR
UNITED STATES LETTERS PATENT

TITLE OF INVENTION:

PROCESSING NETWORK MANAGEMENT DATA IN ACCORDANCE WITH
METADATA FILES

Inventors:

Darryl Black

and

Anne K. Winiewicz

NUTTER, McCLENNEN & FISH, LLP
One International Place
Boston, MA 02110-2699
Telephone (617) 439-2000
Facsimile (617) 973-9748
891632.1

EXPRESS MAIL LABEL NO.: EL327514461US

09637800-081100

EXPRESS MAIL NO. EL327514461US

Docket No. 102689-45/00-U0076

Date of Mailing: August 11, 2000

PROCESSING NETWORK MANAGEMENT DATA
IN ACCORDANCE WITH METADATA FILES

ms B1 *B1* This application is a continuation-in-part of U.S. Serial Number 09/633,675, filed August 7, 2000, entitled "Integrating Operations Support Services With Network Management Systems", still pending.

Background

Periodically accounting data needs to be taken off a network device (e.g., switch, router, hybrid switch-router) and moved to, for example, a workstation for processing and billing integration. Post-processing of the data is necessary to convert it from binary to ASCII, AMA/BAF or other formats. Typically two distributed carefully synchronized processes are used to move the data from the network device to the workstation. If either process becomes out-of-sync with the other (due to a number of factors including power outage, network outage, disk full), data loss is likely to ensue. Data loss may lead to many problems including inaccurate billing – that is, a network provider may be unable to fully bill their customers due to a loss of data showing actual network usage. Data loss may also lead to inaccurate network device performance calculations, which may make it difficult to determine whether quality of service guarantees and service level agreements have been met.

In addition, keeping two or more distributed processes operating in a networked environment is difficult and typically requires one or both processes to maintain the state of the other process. This can add undue burden to the network device. Moreover, network devices have limited storage capacity, and synchronization constraints may cause a network device to exceed its storage capacity leading to data loss and/or a network device crash.

A release of new hardware supporting new file formats requires a new release of software that runs external to the network device and is used to convert the data in accordance with the new file format. In fact, the entire network management system (NMS) software may need to be upgraded and re-released. A new release of software that runs internal to the network device may also be necessary. In either case, the network device and/or network management system may need to be re-booted / re-started in order to begin using the new software.

Summary

The present invention provides a management system internal to a network device that sends various management data files and corresponding metadata files to a management system external to the network device. The external management system then uses the metadata files to process the management data files. As a result, the external management system dynamically learns how to manage a network device through the metadata files. Moreover, new types of management data files -- perhaps corresponding to new hardware within the network device -- may be sent from the internal management system to the external management system along with corresponding new metadata files and the external management system will be able to process the new management files without having to be re-booted or restarted. In addition, multiple network devices coupled with the external management system may send various different types of management data to the external management system and using the metadata files from each network device, the external management system will be able to process the various management data types. In one embodiment, the metadata files are JAVA class files.

In one aspect, the present invention provides a method of operating a telecommunications system including sending a first metadata file from a network device to an external management system, generating a first management data file within the network device, sending the first management data file from the network device to the external management system, and processing the first management data file in accordance with the first metadata file. The first management data file may be generated asynchronously

09637800.08.1.100

or synchronously with respect to the processing of the first management data file, and the first metadata file may be a JAVA class file. Sending the first metadata file and first management data file from the network device to the external management system may include sending the first metadata file and first management data file from the network device to an external file transfer system. The first management data file and/or the first metadata file may be sent to the external management system by executing a file transfer protocol push. The method may also include generating a first data summary file corresponding to the first management data file and sending the first data summary file to the external management system, where the first management data file is processed in accordance with both the first data summary file and the first metadata file. The first data summary file may be sent to the external management system by executing a file transfer protocol push.

The method may also include generating a second management data file within the network device, sending the second management data file from the network device to the external management system and processing the second management data file in accordance with the first metadata file. The method may also include sending a second metadata file from the network device to the external management system, generating a second management data file within the network device, sending the second management data file from the network device to the external management system and processing the second management data file in accordance with the second metadata file. The network device may be a first network device and the method may further include sending a second metadata file from a second network device to the external management system, generating a second management data file within the second network device, sending the second management data file from the second network device to the external management system and processing the second management data file in accordance with the second metadata file. The method may include adding a hardware module to the network device, downloading a second metadata file to the network device corresponding to the hardware module, sending the second metadata file from the network device to the external management system, generating a second management data file within the network device, sending the second management data file from the network device to the

external management system and processing the second management data file in accordance with the second metadata file. In addition, the method may include downloading a modified first metadata file to the network device, sending the modified first metadata file from the network device to the external management system, generating a second management data file within the network device, sending the second management data file from the network device to the external management system and processing the second management data file in accordance with the modified first metadata file. The external management system may be a data collector server, network manager server, billing server or a variety of other types of servers and processes.

In another aspect, the present invention provides a method of operating a telecommunications system including sending a first plurality of metadata files from a first network device to an external management system, generating a first plurality of management data files within the first network device, sending the first management data files from the first network device to the external management system, and processing each of the first management data files in accordance with a corresponding one of the first metadata files. The first management data files may be generated asynchronously or synchronously with respect to the processing of the first management data files, and the first metadata files are JAVA class files. The method may also include sending multiple second metadata files from a second network device to the external management system, generating multiple second management data files within the second network device, sending the second management data files from the second network device to the external management system and processing each of the second management data files in accordance with a corresponding one of the second metadata files. The method may further include adding a hardware module to the first network device, downloading multiple second metadata files to the network device corresponding to the hardware module, sending the second metadata files from the network device to the external management system, generating multiple second management data files within the network device, sending the second management data files from the network device to the external management system, and processing each of the second management data files in accordance with a corresponding one of the second metadata files. The external

Fig. 29 is a block and flow diagram of a computer system incorporating a distributed redundancy architecture and illustrating a method for accomplishing distributed software redundancy;

Fig. 30 is a table representing data in a configuration database;

Figs. 31a-31c, 32a-32c, 33a-33d and 34a-34b are block and flow diagrams of a computer system incorporating a distributed redundancy architecture and illustrating methods for accomplishing distributed redundancy and recovery after a failure;

Fig. 35 is a block diagram of a network device;

Fig. 36 is a block diagram of a portion of a data plane of a network device;

Fig. 37 is a block and flow diagram of a network device incorporating a policy provisioning manager;

Figs. 38 and 39 are tables representing data in a configuration database;

Fig. 40 is an isometric view of a network device;

Figs. 41a-41c are front, back and side block diagrams, respectively, of components and modules within the network device of Fig. 40;

Fig. 42 is a block diagram of dual mid-planes;

Fig. 43 is a block diagram of two distributed switch fabrics and a central switch fabric;

Fig. 44 is a block diagram of the interconnections between switch fabric central timing subsystems and switch fabric local timing subsystems;

Fig. 45 is a block diagram of a switch fabric central timing subsystem;

Fig. 46 is a state diagram of master / slave selection for switch fabric central timing subsystems;

Fig. 47 is a block diagram of a switch fabric local timing subsystem;

Fig. 48 is a state diagram of reference signal selection for switch fabric local timing subsystems;

Fig. 49 is a block diagram of the interconnections between external central timing subsystems and external local timing subsystems;

Fig. 50 is a block diagram of an external central timing subsystem;

Fig. 51 is a timing diagram of a first timing reference signal with an embedded second timing signal;

Fig. 52 is a block diagram of an embeddor circuit;

Fig. 53 is a block diagram of an extractor circuit;

Fig. 54 is a block diagram of an external local timing subsystem;

Fig. 55 is a block diagram of an external central timing subsystem;

Fig. 56 is a block diagram of a network device connected to test equipment through programmable physical layer test ports;

Fig. 57 is a block and flow diagram of a network device incorporating programmable physical layer test ports; and

Fig. 58 is a block diagram of a test path table.

Detailed Description

A modular software architecture solves some of the more common scenarios seen in existing architectures when software is upgraded or new features are deployed. Software modularity involves functionally dividing a software system into individual modules or processes, which are then designed and implemented independently. Inter-process communication (IPC) between the processes is carried out through message passing in accordance with well-defined application programming interfaces (APIs) generated from the same logical system model using the same code generation system. A database process is used to maintain a primary data repository within the computer system / network device, and APIs for the database process are also generated from the same logical system model and using the same code generation system ensuring that all the processes access the same data in the same way. Another database process is used to maintain a secondary data repository external to the computer system / network device; this database receives all of its data by exact database replication from the primary database.

A protected memory feature also helps enforce the separation of modules. Modules are compiled and linked as separate programs, and each program runs in its own protected memory space. In addition, each program is addressed with an abstract communication handle, or logical name. The logical name is location-independent; it can live on any card in the system. The logical name is resolved to a physical card/process during communication. If, for example, a backup process takes over for a failed primary

process, it assumes ownership of the logical name and registers its name to allow other processes to re-resolve the logical name to the new physical card/process. Once complete, the processes continue to communicate with the same logical name, unaware of the fact that a switchover just occurred.

Like certain existing architectures, the modular software architecture dynamically loads applications as needed. Beyond prior architectures, however, the modular software architecture removes significant application dependent data from the kernel and minimizes the link between software and hardware. Instead, under the modular software architecture, the applications themselves gather necessary information (i.e., metadata and instance data) from a variety of sources, for example, text files, JAVA class files and database views, which may be provided at run time or through the logical system model.

Metadata facilitates customization of the execution behavior of software processes without modifying the operating system software image. A modular software architecture makes writing applications – especially distributed applications – more difficult, but metadata provides seamless extensibility allowing new software processes to be added and existing software processes to be upgraded or downgraded while the operating system is running. In one embodiment, the kernel includes operating system software, standard system services software and modular system services software. Even portions of the kernel may be hot upgraded under certain circumstances. Examples of metadata include, customization text files used by software device drivers; JAVA class files that are dynamically instantiated using reflection; registration and deregistration protocols that enable the addition and deletion of software services without system disruption; and database view definitions that provide many varied views of the logical system model. Each of these and other examples are described below.

The embodiment described below includes a network computer system with a loosely coupled distributed processing system. It should be understood, however, that the computer system could also be a central processing system or a combination of distributed and central processing and either loosely or tightly coupled. In addition, the

computer system described below is a network switch for use in, for example, the Internet, wide area networks (WAN) or local area networks (LAN). It should be understood, however, that the modular software architecture can be implemented on any network device (including routers) or other types of computer systems and is not restricted to a network switch.

A distributed processing system is a collection of independent computers that appear to the user of the system as a single computer. Referring to Fig. 1, computer system 10 includes a centralized processor 12 with a control processor subsystem 14 that executes an instance of the kernel 20 including master control programs and server programs to actively control system operation by performing a major portion of the control functions (e.g., booting and system management) for the system. In addition, computer system 10 includes multiple line cards 16a-16n. Each line card includes a control processor subsystem 18a-18n, which runs an instance of the kernel 22a-22n including slave and client programs as well as line card specific software applications. Each control processor subsystem 14, 18a-18n operates in an autonomous fashion but the software presents computer system 10 to the user as a single computer.

Each control processor subsystem includes a processor integrated circuit (chip) 24, 26a-26n, for example, a Motorola 8260 or an Intel Pentium processor. The control processor subsystem also includes a memory subsystem 28, 30a-30n including a combination of non-volatile or persistent (e.g., PROM and flash memory) and volatile (e.g., SRAM and DRAM) memory components. Computer system 10 also includes an internal communication bus 32 connected to each processor 24, 26a-26n. In one embodiment, the communication bus is a switched Fast Ethernet providing 100Mb of dedicated bandwidth to each processor allowing the distributed processors to exchange control information at high frequencies. A backup or redundant Ethernet switch may also be connected to each board such that if the primary Ethernet switch fails, the boards can fail-over to the backup Ethernet switch.

09637800-08100

In this example, Ethernet 32 provides an out-of-band control path, meaning that control information passes over Ethernet 32 but the network data being switched by computer system 10 passes to and from external network connections 31a-31xx over a separate data path 34. External network control data is passed from the line cards to the central processor over Ethernet 32. This external network control data is also assigned a high priority when passed over the Ethernet to ensure that it is not dropped during periods of heavy traffic on the Ethernet.

In addition, another bus 33 is provided for low level system service operations, including, for example, the detection of newly installed (or removed) hardware, reset and interrupt control and real time clock (RTC) synchronization across the system. In one embodiment, this is an Inter-IC communications (I²C) bus.

Alternatively, the control and data may be passed over one common path (in-band).

Network/Element Management System (NMS):

Exponential network growth combined with continuously changing network requirements dictates a need for well thought out network management solutions that can grow and adapt quickly. The present invention provides a massively scalable, highly reliable comprehensive network management system, intended to scale up (and down) to meet varied customer needs.

Within a telecommunications network, element management systems (EMSs) are designed to configure and manage a particular type of network device (e.g., switch, router, hybrid switch-router), and network management systems (NMSs) are used to configure and manage multiple heterogeneous and/or homogeneous network devices. Hereinafter, the term "NMS" will be used for both element and network management systems. To configure a network device, the network administrator uses the NMS to provision services. For example, the administrator may connect a cable to a port of a network device and then use the NMS to enable the port. If the network device supports multiple protocols and services, then the administrator uses the NMS to provision these

as well. To manage a network device, the NMS interprets data gathered by programs running on each network device relevant to network configuration, security, accounting, statistics, and fault logging and presents the interpretation of this data to the network administrator. The network administrator may use this data to, for example, determine when to add new hardware and/or services to the network device, to determine when new network devices should be added to the network, and to determine the cause of errors.

Preferably, NMS programs and programs executing on network devices perform in expected ways (i.e., synchronously) and use the same data in the same way. To avoid having to manually synchronize all integration interfaces between the various programs, a logical system model and associated code generation system are used to generate application programming interfaces (APIs) – that is integration interfaces / integration points -- for programs running on the network device and programs running within the NMS. In addition, the APIs for the programs managing the data repositories (e.g., database programs) used by the network device and NMS programs are also generated from the same logical system model and associated code generation system to ensure that the programs use the data in the same way. Further, to ensure that the NMS and network device programs for managing and operating the network device use the same data, the programs, including the NMS programs, access a single data repository for configuration information, for example, a configuration database within the network device.

Referring to Fig. 2a, in the present invention, the NMS 60 includes one or more NMS client programs 850a-850n and one or more NMS server programs 851a-851n. The NMS client programs provide interfaces for network administrators. Through the NMS clients, the administrator may configure multiple network devices (e.g., computer system 10, Fig. 1; network device 540, Fig. 35). The NMS clients communicate with the NMS servers to provide the NMS servers with configuration requirements from the administrator. In addition, the NMS server provides the NMS client with network device management information, which the client then makes available to the administrator. “Pushing” data from a server to multiple clients synchronizes the clients with minimal polling. Reduced polling means less management traffic on the network and more device CPU cycles

available for other management task. Communication between the NMS client and server is done via Remote Method Invocation (RMI) over Transmission Control Protocol (TCP), a reliable protocol that ensures no data loss.

The NMS client and server relationship prevents the network administrator from directly accessing the network device. Since several network administrators may be managing the network, this mitigates errors that may result if two administrators attempt to configure the same network device at the same time.

The present invention also includes a configuration relational database 42 within each network device and an NMS relational database 61 external to the network device. The configuration database program may be executed by a centralized processor card or a processor on another card (e.g., 12, Fig. 1; 542, Fig. 35) within the network device, and the NMS database program may be executed by a processor within a separate computer system (e.g., 62, Fig. 9a). The NMS server stores data directly in the configuration database via JAVA Database Connectivity (JDBC) over TCP, and using JDBC over TCP, the configuration database, through active queries, automatically replicates any changes to NMS database 61. By using JDBC and a relational database, the NMS server is able to leverage database transactions, database views, database journaling and database backup technologies that help provide unprecedented system availability. Relational database technology also scales well as it has matured over many years. An active query is a mechanism that enables a client to post a blocked SQL query for asynchronous notification by the database when data changes are made after the blocked SQL query was made.

Similarly, any configuration changes made by the network administrator directly through console interface 852 are made to the configuration database and, through active queries, automatically replicated to the NMS database. Maintaining a primary or master repository of data within each network device ensures that the NMS and network device are always synchronized with respect to the state of the configuration. Replicating changes made to the primary database within the network device to any secondary data

repositories, for example, NMS database 61, ensures that all secondary data sources are quickly updated and remain in lockstep synchronization.

Instead of automatically replicating changes to the NMS database through active queries, only certain data, as configured by the network administrator, may be replicated.

Similarly, instead of immediate replication, the network administrator may configure periodic replication. For example, data from the master embedded database (i.e., the configuration database) can be uploaded daily or hourly. In addition to the periodic, scheduled uploads, backup may be done anytime at the request of the network administrator.

Referring again to Fig. 2a, for increased availability, the network device may include a backup configuration database 42' maintained by a separate, backup centralized processor card (e.g., 12, Fig. 1; 543, Fig. 35). Any changes to configuration database 42 are replicated to backup configuration database 42'. If the primary centralized processor card experiences a failure or error, the backup centralized processor card may be switched over to become the primary processor and configuration database 42' may be used to keep the network device operational. In addition, any changes to configuration database 42 may be written immediately to flash persistent memory 853 which may also be located on the primary centralized processor card or on another card, and similarly, any changes to backup configuration database 42' may be written immediately to flash persistent memory 853' which may also be located on the backup centralized processor card or another card. These flash-based configuration files protect against loss of data during power failures. In the unlikely event that all copies of the database within the network device are unusable, the data stored in the NMS database may be downloaded to the network device.

Instead of having a single central processor card (e.g., 12, Fig. 1; 543, Fig. 35), the external control functions and the internal control functions may be separated onto different cards as described in U.S. Patent Application Serial Number ___, filed May 20, 2000 and entitled "Functional Separation of Internal and External Controls in Network

Devices", which is hereby incorporated herein by reference. As shown in Fig. 41a, the chassis may support internal control (IC) processor cards 542a and 543a and external control (EC) processor cards 542b and 543b. In this embodiment, configuration database 42 may be maintained by a processor on internal control processor card 542a and configuration database 42' may be maintained by a processor on internal control processor card 543a, and persistent memory 853 may be located on external control processor card 542b and persistent memory 853' may be located on external control processor card 543b. This increases inter-card communication but also provides increased fault tolerance.

The file transfer protocol (FTP) may provide an efficient, reliable transport out of the network device for data intensive operations. Bulk data applications include accounting, historical statistics and logging. An FTP push (to reduce polling) may be used to send accounting, historical statistics and logging data to a data collector server 857, which may be a UNIX server. The data collector server may then generate network device and/or network status reports 858a-858n in, for example, American Standard Code for Information Interchange (ASCII) format and store the data into a database or generate Automatic Message Accounting Format (AMA/BAF) outputs.

Selected data stored within NMS database 61 may also be replicated to one or more remote/central NMS databases 854a-854n, as described below. NMS servers may also access network device statistics and status information stored within the network device using SNMP (multiple versions) traps and standard Management Information Bases (MIBs and MIB-2). The NMS server augments SNMP traps by providing them over the conventional User Datagram Protocol (UDP) as well as over Transmission Control Protocol (TCP), which provides reliable traps. Each event is generated with a sequence number and logged by the data collector server in a system log database for in place context with system log data. These measures significantly improve the likelihood of responding to all events in a timely manner reducing the chance of service disruption.

The various NMS programs – clients, servers, NMS databases, data collector servers and remote NMS databases – are distributed programs and may be executed on the same computer or different computers. The computers may be within the same LAN or WAN or accessible through the Internet. Distribution and hierarchy are fundamental to making any software system scale to meet larger needs over time. Distribution reduces resource locality constraints and facilitates flexible deployment. Since day-to-day management is done in a distributed fashion, it makes sense that the management software should be distributed. Hierarchy provides natural boundaries of management responsibility and minimizes the number of entities that a management tool must be aware of. Both distribution and hierarchy are fundamental to any long-term management solution. The client server model allows for increased scalability as servers and clients may be added as the number of network managers increase and as the network grows.

The various NMS programs may be written in the JAVA programming language to enable the programs to run on both Windows/NT and UNIX platforms, such as Sun Solaris. In fact the code for both platforms may be the same allowing consistent graphical interfaces to be displayed to the network administrator. In addition to being native to JAVA, RMI is attractive as the RMI architecture includes (RMI) over Internet Inter-Orb Protocol (IIOP) which delivers Common Object Request Broker Architecture (CORBA) compliant distributed computing capabilities to JAVA. Like CORBA, RMI over IIOP uses IIOP as its communication protocol. IIOP eases legacy application and platform integration by allowing application components written in C++, SmallTalk, and other CORBA supported languages to communicate with components running on the JAVA platform. For “manage anywhere” purposes and web technology integration, the various NMS programs may also run within a web browser. In addition, the NMS programs may integrate with Hewlett Packard’s (HP’s) Network Node Manager (NNM™) to provide the convenience of a network map, event aggregation/filtering, and integration with other vendor’s networking. From HP NNM a context-sensitive launch into an NMS server may be executed.

The NMS server also keeps track of important statistics including average client/server response times and response times to each network device. By looking at these statistics over time, it is possible for network administrators to determine when it is time to grow the management system by adding another server. In addition, each NMS server gathers the name, IP address and status of other NMS servers in the telecommunication network, determines the number of NMS clients and network devices to which it is connected, tracks its own operation time, the number of transactions it has handled since initialization, determines the “top talkers” (i.e., network devices associated with high numbers of transactions with the server), and the number of communications errors it has experienced. These statistics help the network administrator tune the NMS to provide better overall management service.

NMS database 61 may be remote or local with respect to the network device(s) that it is managing. For example, the NMS database may be maintained on a computer system outside the domain of the network device (i.e., remote) and communications between the network device and the computer system may occur over a wide area network (WAN) or the Internet. Preferably, the NMS database is maintained on a computer system within the same domain as the network device (i.e., local) and communications between the network device and the computer system may occur over a local area network (LAN). This reduces network management traffic over a WAN or the Internet.

Many telecommunications networks include domains in various geographical locations, and network managers often need to see data combined from these different domains to determine how the overall network is performing. To assist with the management of wide spread networks and still minimize the network management traffic sent over WANs and the Internet, each domain may include an NMS database 61 and particular / selected data from each NMS database may be replicated (or “rolled up”) to remote NMS databases 854a-854n that are in particular centralized locations. Referring to Fig. 2b, for example, a telecommunications network may include at least three LAN domains 855a-855c where each domain includes multiple network devices 540 and an NMS database 61. Domain 855a may be located in the Boston, Massachusetts area, domain 855b may

be located in the Chicago, Illinois area and domain 855c may be located in the San Francisco, California area. NMS servers 851a-851f may be located within each domain or in a separate domain. Similarly, one or more NMS clients may be coupled to each NMS server and located in the same domain as the NMS server or in different domains. In addition, one NMS client may be coupled with multiple NMS servers. For example, NMS servers 851a-851c and NMS clients 850a-850k may be located in domain 856a (e.g., Dallas, Texas) while NMS servers 851d-851f and NMS clients 850m-850u may be located in domain 856b (e.g., New York, New York). Each NMS server may be used to manage each domain 855a-855c or, preferably, one NMS server in each server domain 856a-856b is used to manage all of the network devices within one network device domain 855a-855c. A single domain may include network devices and NMS clients and servers.

Network administrators use the NMS clients to configure network devices in each of the domains through the NMS servers. The network devices replicate changes made to their internal configuration databases (42, Fig. 2a) to a local NMS database 61. In addition, the data collector server copies all logging data into NMS database 61 or a separate logging database (not shown). Each local NMS database may also replicate selected data to central NMS database(s) 854a-854n in accordance with instructions from the network administrator. Other programs may then access the central database to retrieve and combine data from multiple network devices in multiple domains and then present this data to the network administrator. Importantly, network management traffic over WANs and the Internet are minimized since all data is not copied to the central NMS database. For example, local logging data may only be stored in the local NMS databases 61 (or local logging database) and not replicated to one of the central NMS database.

Logical System Model:

As previously mentioned, to avoid having to manually synchronize all integration interfaces between the various programs, the APIs for both NMS and network device programs are generated using a code generation system from the same logical system model. In addition, the APIs for the data repository software used by the programs are

also generated from the same logical system model to ensure that the programs use the data in the same way. Each model within the logical system model contains metadata defining an object / entity, attributes for the object and the object's relationships with other objects. Upgrading / modifying an object is, therefore, much simpler than in current systems, since the relationship between objects, including both hardware and software, and attributes required for each object are clearly defined in one location. When changes are made, the logical system model clearly shows what other programs are affected and, therefore, may also need to be changed. Modeling the hardware and software provides a clean separation of function and form and enables sophisticated dynamic software modularity.

A code generation system uses the attributes and metadata within each model to generate the APIs for each program and ensure lockstep synchronization. The logical model and code generation system may also be used to create test code to test the network device programs and NMS programs. Use of the logical model and code generation system saves development, test and integration time and ensures that all relationships between programs are in lockstep synchronization. In addition, use of the logical model and code generation system facilitates hardware portability, seamless extensibility and unprecedented availability and modularity.

Referring to Fig. 3a, a logical system model 280 is created using the object modeling notation and a model generation tool, for example, Rational Rose 2000 Modeler Edition available from Rational Software Corporation in Lexington, Massachusetts. A managed device 282 represents the top level system connected to models representing both hardware 284 and data objects used by software applications 286. Hardware model 284 includes models representing specific pieces of hardware, for example, chassis 288, shelf 290, slot 292 and printed circuit board 294. The logical model is capable of showing containment, that is, typically, there are many shelves per chassis (1:N), many slots per shelf (1:N) and one board per slot (1:1). Shelf 290 is a parent class generalizing multiple shelf models, including various functional shelves 296a-296n as well as one or more system shelves, for example, for fans 298 and power 300. Board 294 is also a parent

class having multiple board models, including various functional boards without external physical ports 302a-302n (e.g., central processor 12, Fig. 1; 542-543, Fig. 35; and switch fabric cards, Fig. 35) and various functional boards 304a-304n (e.g., cross connection cards 562a-562b and forwarding cards 546a-546e, Fig. 35) that connect to boards 306 with external physical ports (e.g., universal port cards 554a-554h, Fig. 35). Hardware model 284 also includes an external physical port model 308. Port model 308 is coupled to one or more specific port models, for example, synchronous optical network (SONET) protocol port 310, and a physical service endpoint model 312.

Hardware model 284 includes models for all hardware that may be available on computer system 10 (Fig. 1) / network device 540 (Fig. 35) whether a particular computer system / network device uses all the available hardware or not. The model defines the metadata for the system whereas the presence of hardware in an actual network device is represented in instance data. All shelves and slots may not be populated. In addition, there may be multiple classes. It should be understood that SONET port 310 is an example of one type of port that may be supported by computer system 10. A model is created for each type of port available on computer system 10, including, for example, Ethernet, Dense Wavelength Division Multiplexing (DWDM) or Digital Signal, Level 3 (DS3). The NMS (described below) uses the hardware model and instance data to display a graphical picture of computer system 10 / network device 540 to a user.

Service endpoint model 314 spans the software and hardware models within logical model 280. It is a parent class including a physical service endpoint model 312 and a logical service endpoint model 316. Since the links between the software model and hardware model are minimal, either may be changed (e.g., upgraded or modified) and easily integrated with the other. In addition, multiple models (e.g., 280) may be created for many different types of managed devices (e.g., 282). The software model may be the same or similar for each different type of managed device even if the hardware – and hardware models – corresponding to the different managed devices are very different. Similarly, the hardware model may be the same or similar for different managed devices

but the software models may be different for each. The different software models may reflect different customer needs.

Software model 286 includes models of data objects used by each of the software processes (e.g., applications, device drivers, system services) available on computer system 10 / network device 540. All applications and device drivers may not be used in each computer system / network device. As one example, ATM model 318 is shown. It should be understood that software model 286 may also include models for other applications, for example, Internet Protocol (IP) applications, Frame Relay and Multi-Protocol Label Switching (MPLS) applications. Models of other processes (e.g., device drivers and system services) are not shown for convenience.

For each process, models of configurable objects managed by those processes are also created. For example, models of ATM configurable objects are coupled to ATM model 318, including models for a soft permanent virtual path (SPVP) 320, a soft permanent virtual circuit (SPVC) 321, a switch address 322, a cross-connection 323, a permanent virtual path (PVP) cross-connection 324, a permanent virtual circuit (PVC) cross-connection 325, a virtual ATM interface 326, a virtual path link 327, a virtual circuit link 328, logging 329, an ILMI reference 330, PNNI 331, a traffic descriptor 332, an ATM interface 333 and logical service endpoint 316. As described above, logical service endpoint model 316 is coupled to service endpoint model 314. It is also coupled to ATM interface model 333.

The logical model is layered on the physical computer system to add a layer of abstraction between the physical system and the software applications. Adding or removing known (i.e., not new) hardware from the computer system will not require changes to the logical model or the software applications. However, changes to the physical system, for example, adding a new type of board, will require changes to the logical model. In addition, the logical model is modified when new or upgraded processes are created. Changes to an object model within the logical model may require changes to other object models within the logical model. It is possible for the logical

DDL file 348 including SQL commands used to construct various tables and SQL views within a network management (NMS) database 350 (described below). This is also referred to as converting the logical model into a database schema and various SQL views look at particular portions of that schema within the database. If the same database software is used for both the configuration and NMS databases, then one DDL file may be used for both.

The databases do not have to be generated from a logical model for views to work. Instead, database files can be supplied directly without having to generate them using the code generation system. Similarly, instead of using a logical model as an input to the code generation system, a MIB “model” may be used. For example, relationships between various MIBs and MIB objects may be written (i.e., coded) and then this “model” may be used as input to the code generation system.

Referring to Fig. 3c, applications 352a-352n (e.g., SONET driver 863, SONET application 860, MSS 866, etc.) each have an associated view 354a-354n of configuration database 42. The views may be similar allowing each application to view similar data within configuration database 42. For example, each application may be ATM version 1.0 and each view may be ATM view version 1.3. Instead, the applications and views may be different versions. For example, application 352a may be ATM version 1.0 and view 354a may be ATM view version 1.3 while application 352b is ATM version 1.7 and view 354b is ATM view version 1.5. A later version, for example, ATM version 1.7, of the same application may represent an upgrade of that application and its corresponding view allows the upgraded application access only to data relevant to the upgraded version and not data relevant to the older version. If the upgraded version of the application uses the same configuration data as an older version, then the view version may be the same for both applications. In addition, application 352n may represent a completely different type of application, for example, MPLS, and view 354n allows it to have access to data relevant to MPLS and not ATM or any other application. Consequently, through the use of database views, different versions of the same software applications and different types of software applications may be executed on computer system 10 simultaneously.

of each particular network device to which they are connected. Also described below, the NMS servers use the persistent layer metadata as well as run time configuration data to generate SQL configuration commands for use by the configuration database.

Prior to shipping computer system 10 to customers, a software build process is initiated to establish the software architecture and processes. The code generation system is the first part of this process. Following the execution of the code generation system, each process when pulled into the build process links the associated view id and API into its image. For example, referring to Fig. 3d, to build a SONET application, source files, for example, a main application file 859a, a performance monitoring file 859b and an alarm monitoring file 859c, written in, for example, the C programming language (.c) are compiled into object code files (.o) 859a', 859b' and 859c'. Alternatively, the source files may be written in other programming languages, for example, JAVA (.java) or C++ (.cpp). The object files are then linked along with view ids and APIs from the code generation system corresponding to the SONET application, for example, SONET API 340a. The SONET API may be a library (.a) of many object files. Linking these files generates the SONET Application executable file (.exe) 860.

Referring to Fig. 3e, each of the executable files for use by the network device / computer system are then provided to a kit builder 861. For example, several SONET executable files (e.g., 860, 863), ATM executable files (e.g., 864a-864n), MPLS executable files (e.g., 865a-865n), MSS executable files 866a-866n and a DDL configuration database executable file 867 may be provided to kit builder 861. Alternatively, the DDL configuration database executable file may be executed and some data placed in the database prior to supplying the DDL file to the kit builder. The kit builder creates a computer system / network device installation kit 862 that is shipped to the customer with the computer system / network device or, later, alone after modifications and upgrades are made.

Referring to Fig. 3f, similarly, each of the executable files for the NMS is provided separately to the kit builder. For example, a DDL NMS database executable file 868, an

NMS JAVA interfaces executable file 869, a persistent layer metadata executable file 870, an NMS server 885 and an NMS client 886 may be provided to kit builder 861. The kit builder creates an NMS installation kit 871 that is shipped to the customer for installation on a separate computer 62 (Fig. 9a). In addition, new versions of the NMS installation kit may be sent to customers later after upgrades / modifications are made. When installing the NMS, the customer / network administrator may choose to distribute the various NMS processes as described above. Alternatively, one or more of the NMS programs, for example, the NMS JAVA interfaces and Persistent layer metadata executable files may be part of the network device installation kit and later passed from the network device to the NMS server, or part of both the network device installation kit and the NMS installation kit.

When the computer system is powered-up for the first time, as described below, configuration database software uses DDL file 867 to create a configuration database 42 with the necessary configuration tables and active queries. The NMS database software uses DDL file 868 to create NMS database 61 with corresponding configuration tables. Memory and storage space within network devices is typically very limited. The configuration database software is robust and takes a considerable amount of these limited resources but provides many advantages as described below.

As described above, logical model 280 (Fig. 3b) may be provided as an input to code generation system 336 in order to generate database views and APIs for NMS programs and network device programs to synchronize the integration interfaces between those programs. Where a telecommunications network includes multiple similar network devices, the same installation kit may be used to install software on each network device to provide synchronization across the network. Typically, however, networks include multiple different network devices as well as multiple similar network devices. A logical model may be created for each different type of network device and a different installation kit may be implemented on each different type of network device.

Instead, of providing a logical model (e.g., 280, Fig. 3b) that represents a single network device, a logical model may be provided that represents multiple different managed devices – that is, multiple network devices and the relationship between the network devices. Alternatively, multiple logical models 280 and 887a-887n – representing multiple network devices -- may be provided, including relationships with other logical models. In either case, providing multiple logical models or one logical model representing multiple network devices and their relationships as an input(s) to the code generation system allows for synchronization of NMS programs and network device programs (e.g., 901a-901n) across an entire network. The code generation system in combination with one or more logical models provides a powerful tool for synchronizing distributed telecommunication network applications.

The logical model or models may also be used for simulation of a network device and/or a network of many network devices, which may be useful for scalability testing.

In addition to providing view ids and APIs, the code generation system may also provide code used to push data directly into a third party code API. For example, where an API of a third party program expects particular data, the code generation system may provide this data by retrieving the data from the central repository and calling the third-party programs API. In this situation, the code generation system is performing as a “data pump”.

Configuration:

Referring to Fig. 3g, once the network device programs have been installed on network device 540 (Fig. 35), and the NMS programs have been installed on one or more computers (e.g., 62), the network administrator may configure the network device. Since each NMS client may be coupled with many network devices, the administrator begins by using the NMS client to select (step 874) a particular network device to configure. The NMS client then informs (step 875) an NMS server of the particular network device to be configured, and the NMS server using JDBC then connects to the network device and reads the data / table structure from the configuration database within the network device

and uses that information with the JAVA interfaces to construct (step 876) a model of the network device. The server provides (step 877) this information to the client, which displays (step 878) a graphical user interface (GUI) to the administrator indicating the hardware and services available in the selected network device and the current configuration and currently provisioned services.

Referring to Fig. 4a, a GUI 895 may include a graphical depiction of components within a network device. For example, graphic 896 is shown displaying a front view of the components of network device 540 (Fig. 35). A back view and other views may also be shown. The views are used to provide management context; for example, to configure or view statistics about a particular port, the user would select the port from the view and use the left mouse button to bring up the appropriate action on the port. The GUI may also include a configuration / service status window 897 for displaying current configuration and service provisioning details, and a menu 898 for selecting various information to be displayed. The network administrator may configure the network device and provision services through the GUI as well as check logged statistical information.

Fault, Configuration, Accounting, Performance and Security (FCAPS) management are the five functional areas of network management as defined by the International Organization for Standardization (ISO). Fault management is for detecting and resolving network faults, configuration management is for configuring and upgrading the network, accounting management is for accounting and billing for network usage, performance management is for overseeing and tuning network performance, and security management is for ensuring network security. GUI 895 provides a status button 899a-899f for each of the five FCAPS. By clicking on one of the status buttons, a status window appears and displays the status associated with the selected FCAPS button to the network administrator. For example, if the network administrator clicks on the F status button 899a, a fault event summary window 900 (Fig. 4b) appears and displays the status of any faults.

Each FCAP button may be colored according to a hierarchical color code where, for example, green means normal operation, red indicates a serious error and yellow indicates a warning status. Today there are many NMSs that indicate faults through color coded icons or other graphics. However, current NMSs do not categorize the errors or warnings into the ISO five functional areas of network management – that is, FCAPS. The color-coding and order of the FCAPS buttons provide a “status bar code” allowing a network administrator to quickly determine the category of error or warning and quickly take action to address the error or warning.

As with current NMSs, a network administrator may actively monitor the FCAPS buttons by sitting in front of the computer screen displaying the GUI. Unfortunately, network administrators do not have time to actively monitor the status of each network device — passive monitoring is required. To assist passive monitoring, the FCAPS buttons may be enlarged or “stretched” to fill a large portion of the screen, as shown in Fig. 4c. The FCAPS buttons may be stretched in a variety of ways, for example, a stretch option in a pull down menu may be selected or a mouse may be used to drag and drop the borders of the FCAPS buttons. Stretching the FCAPS buttons allows a network administrator to view the status of each FCAP button from a distance of 40 feet or more. Once stretched, each of the five OSI management areas can be easily monitored at a distance by looking at the bar-encoded FCAPS strip. The “stretchy FCAPS” provide instant status recognition at a distance.

The network administrator may set the FCAPS buttons to represent a single network device or multiple network devices or all the network devices in a particular network. Alternatively, the network administrator may have the GUI display two or more FCAPS status bars each of which represents one or more network devices.

Although the FCAPS buttons have been described as a string of multiple stretched bars, many different types of graphics may be used to display FCAPS status. For example, different colors may be used to represent normal operation, warnings and errors, and additional colors may be added to represent particular warnings and/or errors. Instead of

executing the NMS and the network device may go down or the NMS or the network device may crash in the middle of configuring the network device. Current network management systems make configuration changes in a central data repository and pass these changes to network devices using SNMP “sets”. Since changes made through SNMP are committed immediately (i.e., written to the data repository), an uncompleted configuration (series of related “sets”) will leave the network device in a partially configured state (e.g., “dangling” partial configuration records) that is different from the configuration state in the central data repository being used by the NMS. This may cause errors or a network device and/or network failure. To avoid this situation, the configuration database executes groups of SQL commands representing one configuration change as a relational database transaction, such that none of the changes are committed to the configuration database until all commands are successfully executed. The configuration database then notifies the server as to the success or failure of the configuration change. If the server receives a failure notification, then the server re-sends the SQL commands to re-start the configuration changes.

Profiles:

Profiles may be used by the NMS client to provide individual users (e.g., network managers and customers) with customized graphical user interfaces (GUIs) or views of their network and with defined management capabilities. For example, some network managers are only responsible for a certain set of devices in the network. Displaying all network devices makes their management tasks more difficult and may inadvertently provide them with management capabilities over network devices for which they are not responsible or authorized to perform. With respect to customers, profiles limit access to only those network devices in a particular customer's network. This is crucial to protecting the proprietary nature of each customer's network. Profiles also allow each network manager and customer to customize the GUI into a presentation format that is most efficient or easy for them to use. For example, even two users with access to the same network devices and having the same management capabilities may have different GUI customizations through their profiles. In addition, profiles may be used to provide other important information, for example, SNMP community strings to allow an NMS

server to communicate with a network device over SNMP, SNMP retry and timeout values, and which NMS servers to use, for example, primary and secondary servers may be identified.

A network administrator is typically someone who powers up a network device for the first time, installs necessary software on the new network device as well as installs any NMS software on an NMS computer system, and adds any additional hardware and/or software to a network device. The network administrator is also the person that attaches physical network cables to network device ports. The first time GUI 895 is displayed to a network administrator, an NMS client uses a profile including a set of default values. Referring again to Fig. 4a, the administrator may change the default values in his profile by selecting (e.g., clicking on) a profile selection 902 in a navigation tree / menu 898. This causes the NMS client to display a profiles tab 903 (Fig. 4d) on the screen. The profile tab displays any existing profiles 904. The first time the profile tab appears only the network administrator's profile is displayed as no other profiles yet exist.

To save a network manager's time, the profiles tab may also include a copy button 906. By selecting a profile 904 and clicking on the copy button, an existing profile is copied. The network manager may then change the parameters within the copied profile. This is helpful where two user profiles are to include the same or similar parameters.

To change the parameters in the network administrator's profile or any other existing profile, including a copied profile, the user clicks on one of the profiles 904. To add a new profile, the user clicks on an Add button 905. In either case, the NMS client displays a profile dialog box 907 (Fig. 4e) on the screen. Through the profile dialog box, a user's user name 908a, password 908b and confirmed password 908c may be added. The confirm password field is used to assure that the password was entered properly in the password field. The password and confirmed password may be encrypted strings used for user authentication. These fields will be displayed as asterisks on the screen. Once added, a user simply logs on to an NMS client with this user name and password and the NMS client displays the GUI in accordance with the other parameters of this profile.

be set to allow the user to add network device IP addresses to device list field 908g, and a number may be added to a timeout field 908k to specify a number of minutes after which a user will be automatically logged out due to inactivity. A zero in this field or no value in this field may be used to indicate unlimited activity, that is, the user will never be automatically logged out.

The profile may also be used to indicate which NMS servers the NMS client should communicate with. An IP address may be added to a primary server field 908l and a secondary server field 908m. If the primary server fails, the client will access the secondary server. A port number is added to primary server port field 908n and to secondary server port field 908o to indicate the particular ports that should be used for RMI connectivity to the primary and secondary NMS servers.

Additional fields may be added to the device list to provide more information. For example, a read field 908p may be used to indicate the SNMP community string to be used to allow the NMS server to communicate with the network device over SNMP. The SNMP connection may be used to retrieve statistical data from the network device. In addition, a read/write field 908q may be used to indicate an SNMP community string to allow the NMS server to configure the network device and/or provision services. The profile may also include a retry field 908r and a timeout field 908s to provide SNMP retry and timeout values. Many different fields may be provided in a profile.

Instead of providing all the parameters and fields in a single profile dialog box, they may be separated into a variety of a tabbed dialog boxes (Figs. 4f-4i). The tabbed dialog boxes may provide better scalability and flexibility for future needs.

In one embodiment, an administrator level user has both read and write access to the physical and logical objects of the NMS client. Thus, all screens and functionality are available to an administrator level user, and an administrator after physically attaching an

external network attachment to a particular network device port may then enable that port and provision SONET paths on that port. All screens are available to a provisioner level user, however, they do not have access to all functionality as they are limited to read-only access of physical objects. For example, a provisioner can see SONET ports available on a device and can provision SONET paths on a port, but the provisioner cannot enable/disable a SONET port. In other words, a provisioner's power begins at the start of logical objects (not physical objects), for example, SONET paths, ATM interfaces, virtual ATM interfaces, and PVCs, and continues through all the configuration aspects of any object or entity that can be stacked on top of either a SONET path or ATM interface. A customer level user has read-only access to logical entities and only those logical entities corresponding to their group name or listed in the device list field. A customer may or may not have access to Fault, Configuration, Accounting, and Security categories of FCAPS relative to their devices.

A customer may install an NMS client at a customer site or, preferably, the customer will use a web browser to access the NMS client. To use the web browser, a service provider gives the customer an IP address corresponding to the service provider's site. The customer supplies the IP address to their web browser and while at the service provider site, the customer logs in with their username and password. The NMS client then displays the customer level GUI corresponding to that username and password.

Referring to Fig. 4j, a user preference dialog box 909 may be used to customize the GUI into a presentation format that is most efficient or easy for a user to work with. For example, show flags may be used to add tool tips (flag 910a), add horizontal grid lines on tables (flag 910b), add vertical grid lines on tables (flag 910c) and add bookmarks / short cuts (e.g., create a short cut to a PVC dialog box). Look and feel flags may also be used to make the GUI appear as a JAVA GUI would appear (flag 911a) or as a native application, for example, Windows, Windows/NT or Motif, GUI would appear (flag 911b).

Power-Up:

Referring again to Fig. 1, on power-up, reset or reboot, the processor on each board (central processor and each line card) downloads and executes boot-strap code (i.e., minimal instances of the kernel software) and power-up diagnostic test code from its local memory subsystem. After passing the power-up tests, processor 24 on central processor 12 then downloads kernel software 20 from persistent storage 21 into non-persistent memory in memory subsystem 28. Kernel software 20 includes operating system (OS), system services (SS) and modular system services (MSS).

In one embodiment, the operating system software and system services software are the OSE operating system and system services from Enea OSE Systems, Inc. in Dallas, Texas. The OSE operating system is a pre-emptive multi-tasking operating system that provides a set of services that together support the development of distributed applications (i.e., dynamic loading). The OSE approach uses a layered architecture that builds a high level set of services around kernel primitives. The operating system, system services, and modular system services provide support for the creation and management of processes; inter-process communication (IPC) through a process-to-process messaging model; standard semaphore creation and manipulation services; the ability to locate and communicate with a process regardless of its location in the system; the ability to determine when another process has terminated; and the ability to locate the provider of a service by name.

These services support the construction of a distributed system wherein applications can be located by name and processes can use a single form of communication regardless of their location. By using these services, distributed applications may be designed to allow services to transparently move from one location to another such as during a fail over.

The OSE operating system and system services provide a single inter-process communications mechanism that allows processes to communicate regardless of their location in the system. OSE IPC differs from the traditional IPC model in that there are no explicit IPC queues to be managed by the application. Instead each process is assigned

a unique process identification that all IPC messages use. Because OSE IPC supports inter-board communication the process identification includes a path component. Processes locate each other by performing an OSE Hunt call on the process identification. The Hunt call will return the Process ID of the process that maps to the specified path/name. Inter-board communication is carried over some number of communication links. Each link interface is assigned to an OSE Link Handler. The path component of a process path/name is the concatenation of the Link Handler names that one must transverse in order to reach the process.

In addition, the OSE operating system includes memory management that supports a “protected memory model”. The protected memory model dedicates a memory block (i.e., defined memory space) to each process and erects “walls” around each memory block to prevent access by processes outside the “wall”. This prevents one process from corrupting the memory space used by another process. For example, a corrupt software memory pointer in a first process may incorrectly point to the memory space of a second processor and cause the first process to corrupt the second processor’s memory space. The protected memory model prevents the first process with the corrupted memory pointer from corrupting the memory space or block assigned to the second process. As a result, if a process fails, only the memory block assigned to that process is assumed corrupted while the remaining memory space is considered uncorrupted.

The modular software architecture takes advantage of the isolation provided to each process (e.g., device driver or application) by the protected memory model. Because each process is assigned a unique or separate protected memory block, processes may be started, upgraded or restarted independently of other processes.

Referring to Fig. 5, the main modular system service that controls the operation of computer system 10 is a System Resiliency Manager (SRM). Also within modular system services is a Master Control Driver (MCD) that learns the physical characteristics of the particular computer system on which it is running, in this instance, computer system 10. The MCD and the SRM are distributed applications. A master SRM 36 and a

master MCD 38 are executed by central processor 12 while slave SRMs 37a-37n and slave MCDs 39a-39n are executed on each board (central processor 12 and each line card 16a-16n). The SRM and MCD work together and use their assigned view ids and APIs to load the appropriate software drivers on each board and to configure computer system 10.

Also within the modular system services is a configuration service program 35 that downloads a configuration database program 42 and its corresponding DDL file from persistent storage into non-persistent memory 40 on central processor 12. In one embodiment, configuration database 42 is a Polyhedra database from Polyhedra, Inc. in the United Kingdom.

Hardware Inventory and Set-Up:

Master MCD 38 begins by taking a physical inventory of computer system 10 (over the I²C bus) and assigning a unique physical identification number (PID) to each item. Despite the name, the PID is a logical number unrelated to any physical aspect of the component being numbered. In one embodiment, pull-down/pull-up resistors on the chassis mid-plane provide the number space of Slot Identifiers. The master MCD may read a register for each slot that allows it to get the bit pattern produced by these resistors. MCD 38 assigns a unique PID to the chassis, each shelf in the chassis, each slot in each shelf, each line card 16a-16n inserted in each slot, and each port on each line card. (Other items or components may also be inventoried.)

Typically, the number of line cards and ports on each line card in a computer system is variable but the number of chasses, shelves and slots is fixed. Consequently, a PID could be permanently assigned to the chassis, shelves and slots and stored in a file. To add flexibility, however, MCD 38 assigns a PID even to the chassis, shelves and slots to allow the modular software architecture to be ported to another computer system with a different physical construction (i.e., multiple chasses and / or a different number of shelves and slots) without having to change the PID numbering scheme.

Referring to Figs. 5-7, for each line card 16a-16n in computer system 10, MCD 38 communicates with a diagnostic program (DP) 40a-40n being executed by the line card's processor to learn each card's type and version. The diagnostic program reads a line card type and version number out of persistent storage, for example, EPROM 42a-42n, and passes this information to the MCD. For example, line cards 16a and 16b could be cards that implement Asynchronous Transfer Mode (ATM) protocol over Synchronous Optical Network (SONET) protocol as indicated by a particular card type, e.g., 0XF002, and line card 16e could be a card that implements Internet Protocol (IP) over SONET as indicated by a different card type, e.g., 0XE002. In addition, line card 16a could be a version three ATM over SONET card meaning that it includes four SONET ports 44a-44d each of which may be connected to an external SONET optical fiber that carries an OC-48 stream, as indicated by a particular port type 00620, while line card 16b may be a version four ATM over SONET card meaning that it includes sixteen SONET ports 46a-46f each of which carries an OC-3 stream as indicated by a particular port type, e.g., 00820. Other information is also passed to the MCD by the DP, for example, diagnostic test pass / fail status. With this information, MCD 38 creates card table (CT) 47 and port table (PT) 49 in configuration database 42. As described below, the configuration database copies all changes to an NMS database. If the MCD cannot communicate with the diagnostic program to learn the card type and version number, then the MCD assumes the slot is empty.

Even after initial power-up, master MCD 38 will continue to take physical inventories to determine if hardware has been added or removed from computer system 10. For example, line cards may be added to empty slots or removed from slots. When changes are detected, master MCD 38 will update CT 47 and PT 49 accordingly.

For each line card 16a-16n, master MCD 38 searches a physical module description (PMD) file 48 in memory 40 for a record that matches the card type and version number retrieved from that line card. The PMD file may include multiple files. The PMD file includes a table that corresponds card type and version number with name of the mission kernel image executable file (MKI.exe) that needs to be loaded on that line card. Once

determined, master MCD 38 passes the name of each MKI executable file to master SRM 36. Master SRM 36 requests a bootserver (not shown) to download the MKI executable files 50a-50n from persistent storage 21 into memory 40 (i.e., dynamic loading) and passes each MKI executable file 50a-50n to a bootloader (not shown) running on each board (central processor and each line card). The bootloaders execute the received MKI executable file.

Once all the line cards are executing the appropriate MKI, slave MCDs 39a-39n and slave SRMs 37a-37n on each line card need to download device driver software corresponding to the particular devices on each card. Referring to Fig. 8, slave MCDs 39a-39n search PMD file 48 in memory 40 on central processor 12 for a match with their line card type and version number. Just as the master MCD 36 found the name of the MKI executable file for each line card in the PMD file, each slave MCD 39a-39n reads the PMD file to learn the names of all the device driver executable files associated with each line card type and version. The slave MCDs provide these names to the slave SRMs on their boards. Slave SRMs 37a-37n then download and execute the device driver executable files (DD.exe) 56a-56n from memory 40. As one example, one port device driver 43a-43d may be started for each port 44a-44d on line card 16a. The port driver and port are linked together through the assigned port PID number.

In order to understand the significance of the PMD file (i.e., metadata), note that the MCD software does not have knowledge of board types built into it. Instead, the MCD parameterizes its operations on a particular board by looking up the card type and version number in the PMD file and acting accordingly. Consequently, the MCD software does not need to be modified, rebuilt, tested and distributed with new hardware. The changes required in the software system infrastructure to support new hardware are simpler

modify logical model 280 (Fig. 3) to include: a new entry in the PMD file (or a new PMD file) and, where necessary, new device drivers and applications. Because the MCD software, which resides in the kernel, will not need to be modified, the new applications and device drivers and the new DDL files (reflecting the new PMD file) for the

configuration database and NMS database are downloaded and upgraded (as described below) without re-booting the computer system.

Network Management System (NMS):

a Referring to Fig. 9a, as described above, a user / network administrator of computer system 10 works with network management system (NMS) software 60 to configure computer system 10. In the embodiment described below, NMS 60 runs on a personal computer or workstation 62 and communicates with central processor 12 over Ethernet network ~~32~~⁴¹ (out-of-band). Instead, the NMS may communicate with central processor 12 over data path 34 (Fig. 1, in-band). Alternatively (or in addition as a back-up communication port), a user may communicate with computer system 10 through a console interface / terminal (840, Fig. 2a) connected to a serial line 66 connecting to the data or control path using a command line interface (CLI) protocol. Instead, NMS 60 could run directly on computer system 10 provided computer system 10 has an input mechanism for the user.

During installation, an NMS database 61 is established on, for example, work-station 62 using a DDL executable file corresponding to the NMS database. The DDL file may be downloaded from persistent storage 21 in computer system 10 or supplied separately with other NMS programs as part of an NMS installation kit. The NMS database mirrors the configuration database through an active query feature (described below). In one embodiment, the NMS database is an Oracle database from Oracle Corporation in Boston, Massachusetts.

a The NMS and central processor 12 pass control and data over Ethernet ~~32~~⁴¹ using, for example, the Java Database Connectivity (JDBC) protocol. Use of the JDBC protocol allows the NMS to communicate with the configuration database in the same manner that it communicates with its own internal storage mechanisms, including the NMS database. Changes made to the configuration database are passed to the NMS database to ensure that both databases store the same data. This synchronization process is much more efficient, less error-prone and timely than older methods that require the NMS to

periodically poll the network device to determine whether configuration changes have been made. In these systems, NMS polling is unnecessary and wasteful if the configuration has not been changed. Additionally, if a configuration change is made through some other means, for example, a command line interface, and not through the NMS, the NMS will not be updated until the next poll, and if the network device crashes prior to the NMS poll, then the configuration change will be lost. In computer system 10, however, command line interface changes made to configuration database 42 are passed immediately to the NMS database through the active query feature ensuring that the NMS, through both the configuration database and NMS database, is immediately aware of any configuration changes.

Asynchronously Providing Network Device Management Data:

Typically, work-station 62 is coupled to many network computer systems, and NMS 60 is used to configure and manage each of these systems. In addition to configuring each system, the NMS also interprets management data gathered by each system relevant to each system's network accounting data, statistics, security and fault logging and presents this to the user. In current systems, two distributed carefully synchronized processes are used to move data from a network system/device to the NMS. The processes are synchronized with each other by having one or both processes maintain the state of the other process. To avoid the problems associated with using two synchronized processes, in the present invention, internal network device management subsystem processes are made asynchronous with external management processes. That is, neither the internal nor external processes maintain each other's state and all processes operate independently of the other processes. This also minimizes or prevents data loss (i.e., lossless system), which is especially important for revenue generating accounting systems.

In addition, instead of having the NMS interpret each network device's management data in the same fashion, flexibility is added by having each system send the NMS (e.g., data collector server 857, Fig. 2a) class files 410 including compiled source code indicating how its management data should be interpreted. Thus, the NMS effectively "learns" how to process (and perhaps display) management data from the network device via the class

file. Through the reliable File Transfer Protocol (FTP), management subsystem processes 412 running on central processor 12 push data summary files 414 and binary data files 416 to the NMS. Each data summary file indicates the name of the class file the NMS should use to interpret a corresponding binary data file. If the computer system has not already done so, it pushes the class file to the NMS. In one embodiment, the management subsystem processes, class files and NMS processes are JAVA programs, and JAVA Reflection is used to dynamically load the data-specific application class file and process the data in the binary data file. As a result, a new class file can be added or updated on a network device without having to reboot or upgrade the network device or the NMS. The computer system simply pushes the new class file to the NMS. In addition, the NMS can use different class files for each network device such that the data gathered on each device can be particularized to each device.

Referring to Fig. 9b, in one embodiment, the management subsystem 412 (Fig. 9a) is broken into two pieces: a usage data server (UDS) 412a and a file transfer protocol (FTP) client 412b. The UDS is executed on internal processor control card 542a (see also Figs. 41b and 42) while the FTP client is executed on external processor control card 542b (see also Figs. 41a and 42). Alternatively, in a network device with one processor control card or a central processor control card, both the UDS and FTP client may be executed on that one card. When each device driver, for example, SONET driver 415a-415n and ATM driver 417a-417n (only SONET driver 415a and ATM driver 417a are shown for convenience), within network device 540 is built, it links in a usage data monitoring library (UDML). When device drivers are first started, upgraded or re-booted, the UDML causes each device driver to register with the UDS providing one or more string names corresponding to types of data that the device driver will send to the UDS. For example, each ATM driver may register "Acct_PVC" to track permanent virtual circuit statistics, "Acct_SVC" to track soft permanent virtual circuit statistics, "Vir_Intf" to track quality of service (QoS) statistics corresponding to virtual interfaces, and "Bw_Util" to track bandwidth utilization. As another example, each SONET driver may register "Section" to track section statistics, "Line" to track line statistics and "Path" to track path statistics.

Periodically, FTP client 412b asynchronously reads each binary data file and corresponding data summary file from hard drive 421. Preferably, the FTP client reads these files from the hard drive through an out-of-band Ethernet connection, for example, Ethernet 32 (Fig. 1). Alternatively, the FTP client may read these files through an in-band data path 34 (Fig. 1). The FTP client then uses an FTP push to send the binary data file to a file system 425 accessible by the data collector server and, preferably local to the data collector server. The FTP client then uses another FTP push to send the data summary file to the local file system. Since binary data files may be very long and an FTP push of a binary data file may take some time, the data collector server may periodically search the local file system for data summary files. The data collector server may then attempt to open a discovered data summary file. If the data collector server is able to open the file, then that indicates that the FTP push of the data summary file is complete, and since the data summary file is pushed after the binary data file, the data collector server's ability to open the data summary file may be used as an indication that a new binary data file has been completely received. Since data summary files are much smaller than binary data files, having the data collector server look for and attempt to open data summary files instead of binary data files minimizes the thread wait within the data collector server.

In one embodiment, the data collector server is a JAVA program, and each different type of binary data file has a corresponding JAVA class file (e.g., class file 410a) that defines how the data collector server should process the binary data file. When a device driver is loaded into the network device, a corresponding JAVA class file is also loaded and stored in hard drive 421. The FTP client periodically polls the hard drive for new JAVA class files and uses an FTP push to send them to file system 425. The data collector server uses the binary file type in the data summary file to determine which JAVA class file it should use to interpret the binary data file. The data collector server then converts the binary data into ASCII or AMA/BAF format and stores the ASCII or AMA/BAF files in the file system. The data collector server may use a set of worker threads for concurrency.

As described, the data collector server is completely independent of and asynchronous with the FTP client, which is also independent and asynchronous of the UDS. The separation of the data collector server and FTP client avoids data loss due to process synchronization problems, since there is no synchronization, and reduces the burden on the network device by not requiring the network device to maintain synchronization between the processes. In addition, if the data collector server goes down or is busy for some time, the FTP client and UDS continue working and continue sending binary data files and data summary files to the file system. When the data collector server is again available, it simply accesses the data summary files and processes the binary files as described above. Thus, there is no data loss and the limited storage capacity within the network device is not strained by storing data until the data collector server is available. In addition, if the FTP client or UDS goes down, the data collector server may continue working.

An NMS server (e.g., NMS server 851a), which may or may not be executing on the same computer system 62 as the data collector server, may periodically retrieve the ASCII or AMA/BAF files from the file system. The files may represent accounting, statistics, security, logging and/or other types of data gathered from hardware within the network device. The NMS server may also access the corresponding class files from the file system to learn how the data should be presented to a user, for example, how a graphical user interface (GUI) should be displayed, what data and format to display, or perhaps which one of many GUIs should be used. The NMS server may use the data to, for example, monitor network device performance, including quality of service guarantees and service level agreements, as well as bill customers for network usage. Alternatively, a separate billing server 423a or statistics server 423b, which may or may not be executing on the same computer system 62 as the data collector server and/or the NMS server, may periodically retrieve the ASCII or AMA/BAF files from the file system in order to monitor network device performance, including quality of service guarantees and service level agreements, and/or bill customers for network usage. One or more of the data collector server, the NMS server, the billing server and the statistics server may be combined into one server. Moreover, management files created by the data collector

server may be combined with data from the configuration or NMS databases to generate billing records for each of the network provider's customers.

The data collector server may convert the ASCII or AMA/BAF files into other data formats, for example, Excel spread sheets, for use by the NMS server, billing server and/or statistics server. In addition, the application class file for each data type may be modified to go beyond conversion, including direct integration into a database or an OSS system. For example, many OSS systems use a Portal billing system available from Portal Software, Inc. in Cupertino, CA. The JAVA class file associated with a particular binary data file and data summary file may cause the data collector server to convert the binary data file into ASCII data and then issue a Portal API call to give the ASCII data directly to the Portal billing system. As a result, accounting, statistics, logging and/or security data may be directly integrated into any other process, including third party processes, through JAVA class files.

Through JAVA class files, new device drivers may be added to a network device without having to change UDS 412a or FTP client 412b and without having to re-boot the network device and without having to upgrade/modify external processes. For example, a new forwarding card (e.g., forwarding card 552a) may be added to an operating network device and this new forwarding card may support MPLS. An MPLS device driver 419, linked within the UDML, is downloaded to the network device as well as a corresponding class file (e.g., class file 410e). When the FTP client discovers the new class file in hard drive 421, it uses an FTP push to send it to file system 425. The FTP client does not need to understand the data within the class file it simply needs to push it to the file system. Just as with other device drivers, the UDML causes the MPLS driver to register appropriate string names with the UDS and poll and send data to the UDS with a registered string name. The UDS stores binary data files (e.g., binary data file 416e) and corresponding data summary files (e.g., data summary file 414e) in the hard drive without having to understand the data within the binary data file. The FTP client then pushes these files to the file system again without having to understand the data. When the data summary file is discovered by the data collector server, the data collector server

09637800-031100

uses the binary file type in the data summary file to locate the new MPLS class file 410e in the file system and then uses the class file to convert the binary data in the corresponding binary data file into ASCII format and perhaps other data formats. Thus, a new device driver is added and statistical information may be gathered without having to change any of the other software and without having to re-boot the network device.

As described, having the data collector server be completely independent of and asynchronous with the FTP client avoids the typical problems encountered when internal and external management programs are synchronized. Moreover, modularity of device drivers and internal management programs is maintained by providing metadata through class files that instruct the external management programs as to how the management data should be processed. Consequently, device drivers may be modified, upgraded and added to an operating network device without disrupting the operation of any of the other device drivers or the management programs.

Configuration:

As described above, unlike a monolithic software architecture which is directly linked to the hardware of the computer system on which it runs, a modular software architecture includes independent applications that are significantly decoupled from the hardware through the use of a logical model of the computer system. Using the logical model and a code generation system, a view id and API are generated for each application to define each application's access to particular data in a configuration database and programming interfaces between the different applications. The configuration database is established using a data definition language (DDL) file also generated by the code generation system from the logical model. As a result, there is only a limited connection between the computer system's software and hardware, which allows for multiple versions of the same application to run on the computer system simultaneously and different types of applications to run simultaneously on the computer system. In addition, while the computer system is running, application upgrades and downgrades may be executed without affecting other applications and new hardware and software may be added to the system also without affecting other applications.

Referring again to Fig. 9a, initially, NMS 60 reads card table 47 and port table 49 to determine what hardware is available in computer system 10. The NMS assigns a logical identification number (LID) 98 (Figs. 11a and 11b) to each card and port and inserts these numbers in an LID to PID Card table (LPCT) 100 and an LID to PID Port table (LPPT) 101 in configuration database 42. Alternatively, the NMS could use the PID previously assigned to each board by the MCD. However, to allow for hardware redundancy, the NMS assigns an LID and may associate the LID with at least two PIDs, a primary PID 102 and a backup PID 104. (LPCT 100 may include multiple backup PID fields to allow more than one backup PID to be assigned to each primary PID.)

The user chooses the desired redundancy structure and instructs the NMS as to which boards are primary boards and which boards are backup boards. For example, the NMS may assign LID 30 to line card 16a -- previously assigned PID 500 by the MCD -- as a user defined primary card, and the NMS may assign LID 30 to line card 16n -- previously assigned PID 513 by the MCD -- as a user defined back-up card (see row 106, Fig. 11a). The NMS may also assign LID 40 to port 44a -- previously assigned PID 1500 by the MCD -- as a primary port, and the NMS may assign LID 40 to port 68a -- previously assigned PID 1600 by the MCD -- as a back-up port (see row 107, Fig. 11b).

In a 1:1 redundant system, each backup line card backs-up only one other line card and the NMS assigns a unique primary PID and a unique backup PID to each LID (no LIDs share the same PIDs). In a 1:N redundant system, each backup line card backs-up at least two other line cards and the NMS assigns a different primary PID to each LID and the same backup PID to at least two LIDs. For example, if computer system 10 is a 1:N redundant system, then one line card, for example, line card 16n, serves as the hardware backup card for at least two other line cards, for example, line cards 16a and 16b. If the NMS assigns an LID of 31 to line card 16b, then in logical to physical card table 100 (see row 109, Fig. 11a), the NMS associates LID 31 with primary PID 501 (line card 16b) and backup PID 513 (line card 16n). As a result, backup PID 513 (line card 16n) is associated with both LID 30 and 31.

00000000000000000000

Service endpoint managers (SEMs) within the modular system services of the kernel software running on each line card use the service endpoint numbers assigned by the NMS to enable ports and to link instances of applications, for example, ATM, running on the line cards with the correct port. The kernel may start one SEM to handle all ports on one line card, or, for resiliency, the kernel may start one SEM for each particular port. For example, SEMs 96a-96d are spawned to independently control ports 44a-44d.

The service endpoint managers (SEMs) running on each board establish active queries with the configuration database for SET 76. Thus, when the NMS changes or adds to the service endpoint table (SET), the configuration database sends the service endpoint manager associated with the port PID in the SET a change notification including information on the change that was made. In the continuing example, configuration database 42 notifies SEM 96a that SET 76 has been changed and that SE 1 was assigned to port 44a (PID 1500). Configuration database 42 notifies SEM 96b that SE 2, 3, and 4 were assigned to port 44b (PID 1501), SEM 96c that SE 5 and 6 were assigned to port 44c (PID 1502) and SEM 96d that SE 7, 8, and 9 were assigned to port 44d (PID 1503). When a service endpoint is assigned to a port, the SEM associated with that port passes the assigned SE number to the port driver for that port using the port PID number associated with the SE number.

To load instances of software applications on the correct boards, the NMS creates software load records (SLR) 128a-128n in configuration database 42. The SLR includes the name 130 (Fig. 14) of a control shim executable file and an LID 132 for cards on which the application must be spawned. In the continuing example, NMS 60 creates SLR 128a including the executable name `atm_cntrl.exe` and card LID 30 (row 134). The configuration database detects LID 30 in SLR 128a and sends slave SRMs 37b (line card 16a) and 37o (line card 16n) a change notification including the name of the executable file (e.g., `atm_cntrl.exe`) to be loaded. The primary slave SRMs then download and execute a copy of `atm_cntrl.exe` 135 from memory 40 to spawn the ATM controllers (e.g., ATM controller 136 on line card 16a). Since slave SRM 37o is on backup line card 16n, it may or may not spawn an ATM controller in backup mode. Software backup is described in more detail below. Instead of downloading a copy of `atm_cntrl.exe` 135 from memory 40, a slave SRM may download it from another line card that already downloaded a copy from memory 40. There may be instances when downloading from a line card is quicker than downloading from central processor 12. Through software load records and the tables in configuration database 42, applications are downloaded and executed without the need for the system services, including the SRM, or any other software in the kernel to have information as to how the applications should be

configured. The control shims (e.g., atm_cntrl.exe 135) interpret the next layer of the application (e.g., ATM) configuration.

For each application that needs to be spawned, for example, an ATM application and a SONET application, the NMS creates an application group table. Referring to Fig. 12, ATM group table 108 indicates that four instances of ATM (i.e., group number 1, 2, 3, 4) – corresponding to four enabled ports 44a-44n -- are to be started on line card 16a (LID 30). If other instances of ATM are started on other line cards, they would also be listed in ATM group table 108 but associated with the appropriate line card LID. ATM group table 108 may also include additional information needed to execute ATM applications on each particular line card. (See description of software backup below.)

In the above example, one instance of ATM was started for each port on the line card. This provides resiliency and fault isolation should one instance of ATM fail or should one port suffer a failure. An even more resilient scheme would include multiple instances of ATM for each port. For example, one instance of ATM may be started for each path received by a port.

The application controllers on each board now need to know how many instances of the corresponding application they need to spawn. This information is in the application group table in the configuration database. Through the active query feature, the configuration database notifies the application controller of records associated with the board's LID from corresponding application group tables. In the continuing example, configuration database 42 sends ATM controller 136 records from ATM group table 108 that correspond to LID 30 (line card 16a). With these records, ATM controller 136 learns that there are four ATM groups associated with LID 30 meaning ATM must be instantiated four times on line card 16a. ATM controller 136 asks slave SRM 37b to download and execute four instances (ATM 110-113, Fig. 15) of atm.exe 138.

Once spawned, each instantiation of ATM 110-113 sends an active database query to search ATM interface table 114 for its corresponding group number and to retrieve

associated records. The data in the records indicates how many ATM interfaces each instantiation of ATM needs to spawn. Alternatively, a master ATM application (not shown) running on central processor 12 may perform active queries of the configuration database and pass information to each slave ATM application running on the various line cards regarding the number of ATM interfaces each slave ATM application needs to spawn.

Referring to Figs. 13 and 15, for each instance of ATM 110-113 there may be one or more ATM interfaces. To configure these ATM interfaces, the NMS creates an ATM interface table 114. There may be one ATM interface 115-122 per path / service endpoint or multiple virtual ATM interfaces 123-125 per path. This flexibility is left up to the user and NMS, and the ATM interface table allows the NMS to communicate this configuration information to each instance of each application running on the different line cards. For example, ATM interface table 114 indicates that for ATM group 1, service endpoint 1, there are three virtual ATM interfaces (ATM-IF 1-3) and for ATM group 2, there is one ATM interface for each service endpoint: ATM-IF 4 and SE 2; ATM-IF 5 and SE 3; and ATM-IF 6 and SE 4.

Computer system 10 is now ready to operate as a network switch using line card 16a and ports 44a-44d. The user will likely provide the NMS with further instructions to configure more of computer system 10. For example, instances of other software applications, such as an IP application, and additional instances of ATM may be spawned (as described above) on line cards 16a or other boards in computer system 10.

As shown above, all application dependent data resides in memory 40 and not in kernel software. Consequently, changes may be made to applications and configuration data in memory 40 to allow hot (while computer system 10 is running) upgrades of software and hardware and configuration changes. Although the above described power-up and configuration of computer system 10 is complex, it provides massive flexibility as described in more detail below.

Template Driven Service Provisioning:

Instead of using the GUI to interactively provision services on one network device in real time, a user may provision services on one or more network devices in one or more networks controlled by one or more network management systems (NMSs) interactively and non-interactively using an Operations Support Services (OSS) client and templates. At the heart of any carrier's network is the OSS, which provides the overall network management infrastructure and the main user interface for network managers / administrators. The OSS is responsible for consolidating a diverse set of element/network management systems and third-party applications into a single system that is used, for example, to detect and resolve network faults (Fault Management), configure and upgrade the network (Configuration Management), account and bill for network usage (Accounting Management), oversee and tune network performance (Performance Management), and ensure ironclad network security (Security Management). FCAPS are the five functional areas of network management as defined by the International Organization for Standardization (ISO). Through templates one or more NMSs may be integrated with a telecommunication network carrier's OSS.

Templates are metadata and include scripts of instructions and parameters. In one embodiment, instructions within templates are written in ASCII text to be human readable. There are three general categories of templates, provisioning templates, control templates and batch templates. A user may interactively connect the OSS client with a particular NMS server and then cause the NMS server to connect to a particular device. Instead, the user may create a control template that non-interactively establishes these connections. Once the connections are established, whether interactively or non-interactively, provisioning templates may be used to complete particular provisioning tasks. The instructions within a provisioning template cause the OSS client to issue appropriate calls to the NMS server which cause the NMS server to complete the provisioning task, for example, by writing / modifying data within the network device's configuration database. Batch templates may be used to concatenate a series of templates and template modifications (i.e., one or more control and provisioning templates) to provision one or more network devices. Through the client / server based architecture,

multiple OSS clients may work with one or more NMS servers. Database view ids and APIs for the OSS client may be generated using the logical model and code generation system (Fig. 3b) to synchronize the integration interfaces between the OSS clients and the NMS servers.

Interactively, a network manager may have an OSS client execute many provisioning templates to complete many provisioning tasks. Instead, the network manager may order and sequence the execution of many provisioning templates within a batch template to non-interactively complete the many provisioning tasks and build custom services. In addition, execution commands followed by control template names may be included within batch templates to non-interactively cause an OSS client to establish connections with particular NMS servers and network devices. For example, a first control template may designate a network device to which the current OSS client and NMS server are not connected. Including an execution command followed by the first control template name in a batch template will cause the OSS client to issue calls to the NMS server to cause the NMS server to access the different network device. As another example, a second control template may designate an NMS server and a network device to which the OSS client is not currently connected. Including an execution command followed by the second control template name will cause the OSS client to set up connections to both the different NMS server and the different network device. Moreover, batch templates may include execution commands followed by provisioning template names after each execution command and control template to provision services within the network devices designated by the control templates. Through batch templates, therefore, multiple control templates and provisioning templates may be ordered and sequenced to provision services within multiple network devices in multiple networks controlled by multiple NMSs.

Calls issued by the OSS client to the NMS server may cause the NMS server to immediately provision services or delay provisioning services until a predetermined time, for example, a time when the network device is less likely to be busy. Templates may be written to apply to different types of network devices.

A “command line” interactive interpreter within the OSS client may be used by a network manager to select and modify existing templates or to create new templates. Templates may be generated for many various provisioning tasks, for example, setting up a permanent virtual circuit (PVC), a switched virtual circuit (SVC), a SONET path (SPATH), a traffic descriptor (TD) or a virtual ATM interface (VAIF). Once a template is created, a network manager change default parameters within the template to complete particular provisioning tasks. A network manager may also copy a template and modify it to create a new template.

Referring to Fig. 3h, using the interactive interpreter, a network administrator may provision services by selecting (step 888) a template and using the default parameters within that template or copying and renaming (step 889) a particular provisioning template corresponding to a particular provisioning task and either accepting default parameter values provided by the template or changing (step 890) those default values to meet the administrator’s needs. The network administrator may also change parameters and instructions within a copy of a template to create a new template. The modified provisioning templates are sent to or loaded into (step 891) the OSS client, which executes the instructions within the template and issues the appropriate calls (step 892) to the NMS server to satisfy the provisioning need. The OSS client may be written in JAVA and employ script technology. In response to calls received from the OSS client, the NMS server may execute (step 894) the provisioning requests defined by a template immediately or in a “batch-mode” (step 893), perhaps with other calls received from the OSS client or other clients, at a time when network transactions are typically low (e.g., late at night).

Referring to Fig. 3i, at the interactive interpreter prompt 912 (e.g., Enetcli>) a network manager may type in “help” and be provided with a list (e.g., list 913) of commands that are available. In one embodiment, available commands may include bye, close, execute, help, load, manage, open, quit, showCurrent, showTemplate, set, status, writeCurrent, and writeTemplate. Many different commands are possible. The bye command allows

the network manager to exit the interactive interpreter, the close command allows the network manager to close a connection between the OSS client and that NMS server, and the execute command followed by a template type causes the OSS client to execute the instructions within the loaded template corresponding to that template type.

As shown, the help command alone causes the interactive interpreter to display the list of commands. The help command followed by another command provides help information about that command. The load command followed by a template type and a named template loads the named template into the OSS client such that any commands followed by the template type will use the named/loaded template. The manage command followed by an IP address of a network device causes the OSS client to issue a call to an NMS server to establish a connection between the NMS server and that network device. Alternatively, a username and password may also need to be supplied. The open command followed by an NMS server IP address causes the OSS client to open a connection with that NMS server, and again, the network manager may also need to supply a username and password. Instead of an IP address, a domain name server (DNS) name may be provided and a host look up may be used to determine the IP address and access the corresponding device.

The `showCurrent` command followed by a template type will cause the interactive interpreter to display current parameter values for the loaded template corresponding to that template type. For example, `showCurrent SPATH 914` displays a list 915 of parameters and current parameter values for the loaded template corresponding to the SPATH template type. The `showTemplate` command followed by a template type will cause the OSS client to display available parameters and acceptable parameter values for each parameter within the loaded template. For example, `showTemplate SPATH 916` causes the interactive interpreter to display the available parameters 917 within the loaded template corresponding to the SPATH template type. The `set` command followed by a template type, a parameter name and a value will change the named parameter to the designated value within the loaded template, and a subsequent `showCurrent` command followed by that template type will show the new parameter value within the loaded.

The status command 918 will cause the interactive interpreter to display a status of the current interactive interpreter session. For example, the interactive interpreter may display the name 919 of an NMS server to which the OSS client is currently connected (as shown in Fig. 3i, the OSS client is currently not connected to an NMS server) and the interactive interpreter may display the names 920 of available template types. The writeCurrent command followed by a template type and a new template name will cause the interactive interpreter to make a copy of the loaded template, including current parameter values, with the new template name. The writeTemplate command followed by a template type and a new template name, will cause the interactive interpreter to make a copy of the template with the new template name with placeholders values (i.e., <String>) that indicate the network manager needs to fill in the template with the required datatypes as parameter values. The network manager may then use the load command followed by the new template name to load the new template into the OSS client.

Referring to Fig. 3j, from the interactive interpreter prompt (e.g., Enetcli>), a network manager may interactively provision services on a network device. The network manager begins by typing an open command 921a followed by the IP address of an NMS server to cause the OSS client to open a connection 921b with that NMS server. The network manager may then issue a manage command 921c followed by the IP address of a particular network device to cause the OSS client to issue a call 921d to the NMS server to cause the NMS server to open a connection 921e with that network device.

The network manager may now provision services within that network device by typing in an execute command 921f followed by a template type. For example, the network manager may type "execute SPATH" at the Enetcli> prompt to cause the OSS client to execute the instructions 921g within the loaded SPATH template using the parameter values within the loaded SPATH template. Executing the instructions causes the OSS client to issue calls to the NMS server, and these calls cause the NMS server to complete the provisioning task 921h. For example, following an execute SPATH command, the

09637800-081100

NMS server will set up a SONET path in the network device using the parameter values passed to the NMS server by the OSS client from the template.

At any time from the Enetcli> prompt, a network manager may change the parameter values within a template. Again, the network manager may use showCurrent followed by a template type to see the current parameter values within the loaded template or showTemplate to see the available parameters within the loaded template. The network manager may then use the set command followed by the template type, parameter name and new parameter value to change a parameter value within the loaded template. For example, after the network manager sets up a SONET path within the network device, the network manager may change one or more parameter values within the loaded SPATH template and re-execute the SPATH template to set up a different SONET path within the same network device.

Once a connection to a network device is open, the network manager may interactively execute any template any number of times to provision services within that network device. The network manager may also create new templates and execute those. The network manager may simply write a new template or use the writeCurrent or writeTemplate commands to copy an existing template into a new template name and then edit the instructions within the new template.

After provisioning services within a first network device, the network manager may open a connection with a second network device to provision services within that second network device. If the NMS server currently connected to the OSS client is capable of establishing a connection with the second network device, then the network manager may simply open a connection to the second network device. If the NMS server currently connected to the OSS client is not capable of establishing a connection with the second network device, then the network manager closes the connections with the NMS server and then opens connections with a second NMS server and the second network device. Thus, a network manager may easily manage / provision services within multiple network devices within multiple networks even if they are managed by different NMS

servers. In addition, other network managers may provision services on the same network devices through the same NMS servers using other OSS clients that are perhaps running on other computer systems. That is, multiple OSS clients may be connected to multiple NMS servers.

Instead of interactively establishing connections with NMS servers and network devices, control templates may be used to non-interactively establish these connections. Referring to Fig. 3k, using a showCurrent command 922 followed by CONTROL causes the interactive interpreter to display parameters available in the loaded CONTROL template. In one embodiment, an execute control command will automatically cause the OSS client to execute instructions within the loaded CONTROL template and open a connection to an NMS server designated within the CONTROL template. Since the OSS client automatically opens a connection with the designated NMS server, the open command may but need not be included within the CONTROL template. In this example, the CONTROL template includes "localhost" 923a as the DNS name of the NMS server with which the OSS client should open a connection. In one embodiment, "localhost" refers to the same system as the OSS client. A username 923b and password 923c may also need to be used to open the connection with the localhost NMS server. The CONTROL template also includes the manage command 923d and a network device IP address 923e of 192.168.9.202. With this information (and perhaps the username and password or another username and password), the OSS client issues calls to the localhost NMS server to cause the server to set up a connection with that network device.

The template may also include an output file name 923f where any output / status information generated in response to the execution of the CONTROL template will be sent. The template may also include a version number 923g. Version numbers allow a new template to be created with the same name as an old template but with a new version number, and the new template may include additional/different parameters and/or instructions. Using version numbers, both old (e.g., not upgraded) and new OSS clients may use the templates but only access those templates having particular version numbers that correspond to the functionality of each OSS client.

Once connections with an NMS server and network device are established (either interactively or non-interactively through a control template), services within the network device may be provisioned. As described above, a network manager may interactively provision services by issuing execute commands followed by provisioning template types. Alternatively, a network manager may provision services non-interactively through batch templates, which include an ordered list of tasks, including execute commands followed by provisioning template types.

Referring to Fig. 3L, a batch template type named BATCH 924 includes an ordered list of tasks, including execute commands followed by provisioning template types. When a network manager issues an execute command followed by the BATCH template type at the Enetcli> prompt, the OSS client will carry out each of the tasks within the loaded BATCH template. In this example, task1 924a includes “execute SPATH” which causes the OSS client to establish a SONET path within the network device to which a connection is open, task2 924b includes “execute PVC” to cause the OSS client to set up a permanent virtual circuit within the network device, and task3 924c includes “execute SPVC” to cause the OSS client to set up a soft permanent virtual circuit within the network device.

If multiple similar provisioning tasks are needed, then the network manager may use writeCurrent or writeTemplate to create multiple similar templates (i.e., same template type with different template names), change or add parameter values within these multiple similar templates using the set command, and sequentially load and execute each of the different named templates. For example, SPVC is the template type and task3 causes the OSS to execute instructions within the previously loaded named template. Spvc1 and spvc2 are two different named templates (or template instantiations) corresponding to the SPVC template type for setting up soft permanent virtual circuits having different parameters from each other and the loaded template to set up different SPVCs. In this example, the BATCH template then includes task4 924d including “load SPVC spvc1” to load the spvc1 template and then task5 924e “execute SPVC” to cause

the OSS client to execute the loaded spvc1 template and set up a different SPVC. Similarly, task6 924f includes “load SPVC spvc2” and task7 924e includes “execute SPVC” to cause the OSS client to execute the loaded spvc2 template and set up yet another different SPVC.

Alternatively, the batch template may include commands for altering an existing template such that multiple similar templates are not necessary. For example, the loaded BATCH template may include task50 924g “set SPATH PortID 3” to cause the OSS client to change the PortID parameter within the SPATH template to 3. The BATCH template then includes task51 924h “execute SPATH” 924g to cause the OSS client to execute the SPATH template including the new parameter value which sets up a different SONET path. A BATCH template may include many set commands to change parameter values followed by execute commands to provision multiple similar services within the same network device. For example, the BATCH template may further include task52 924i “set SPATH SlotID 2” followed by task53 924j “execute SPATH” to set up yet another different SONET path. Using this combination of set and execute commands eliminates the need to write, store and keep track of multiple similar templates.

Batch templates may also be used to non-interactively provision services within multiple different network devices by ordering and sequencing tasks including execute commands followed by control template types and then execute commands followed by provisioning template types. Referring to Fig. 3M, instead of non-interactively establishing connections with an NMS server and a network device using a control template, a batch template may be used. For example, the first task in a loaded BATCH template 925 may be task1 925a “execute CONTROL”. This will cause the OSS client to execute the loaded CONTROL template to establish connections with the NMS server and the network device designated within the loaded CONTROL template (e.g., localhost and 192.168.9.202). The BATCH template then includes provisioning tasks, for example, task2 925b includes “execute SPATH” to set up a SONET path, and task3 925c includes “set SPATH PortID 3” and task4 925d includes “execute SPATH” to set up a different

SONET path. Many additional provisioning tasks for this network device may be completed in this way.

The BATCH template may then have a task including a set command to modify one or more parameters within a control template to cause the OSS client to set up a connection with a different network device and perhaps a different NMS server. Where the network manager wishes to provision a network device capable of being connected to through the currently connected NMS server, for example, localhost, then the BATCH template need only have task61 925e including “set CONTROL System” followed by the IP address of the different network device, for example, 192.168.9.201. The BATCH template then has a task62 925f including “execute CONTROL”, which causes the OSS client to issue calls to the localhost NMS server to establish a connection with the different network device. The BATCH template may then have tasks including execute commands followed by provisioning templates, for example, task63 925g including “execute SPATH”, to provision services within the different network device.

If the network manager wishes to provision a network device coupled with another NMS server, then the BATCH template includes, for example, task108 925h including “close” to drop the connection between the OSS client and localhost NMS server. The BATCH template may then have, for example, task109 925i including “set CONTROL Server Server1” to change the server parameter within the loaded CONTROL template to Server1 and task110 925j including “set CONTROL System 192.168.8.200” to change the network device parameter within the loaded CONTROL template to the IP address of the new network device. The BATCH template may then have task111 925k including “execute CONTROL” to cause the OSS client to set up connections to the Server1 NMS server and to network device 192.168.8.200. The BATCH template may then include tasks with execute commands followed by provisioning template types to provision services within the network device, for example, task112 925L includes “execute SPATH”.

09637800-031100

The templates and interactive interpreter / OSS client may be loaded and executed on a central OSS computer system(s) and used to provision services in one or more network devices in one or more network domains. A network administrator may install an OSS client at various locations and/or for “manage anywhere” purposes, web technology may be used to allow a network manager to download an OSS client program from a web accessible server onto a computer at any location. The network manager may then use the OSS client in the same manner as when it is loaded onto a central OSS computer system. Thus, the network manager may provision services from any computer at any location.

Provisioning templates may be written to apply to different types of network devices. The network administrator does not need to know details of the network device being provisioned as the parameters required and available for modification are listed in the various templates. Consequently, the templates allow for multifaceted integration of different network management systems (NMS) into existing OSS infrastructures.

Instead of using template executable files and an OSS client, network managers may prefer to use their standard OSS interface to provision services in various network devices. In one embodiment, therefore, a single OSS client application programming interface (API) and a library of compiled code may be linked directly into the OSS software. The library of compiled code is a subset of the compiled code used to create the OSS client, with built-in templates including provisioning, control, batch and other types of templates. The OSS software then uses the supported templates as documentation of the necessary parameters needed for each provisioning task and presents template streams (null terminated arrays of arguments that serialize the totality of arguments required to construct a supported template) via the single API for potential alteration through the OSS standard interface. Since the network managers are comfortable working with the OSS interface, provisioning services may be made more efficient and simple by directly linking the OSS client API and templates into the OSS software.

Typically, OSS software is written in C or C++ programming language. In one embodiment, the OSS client and templates are written in JAVA, and JAVA Native Interface (JNI) is used by the OSS software to access the JAVA OSS client API and templates.

Inter-Process Communication:

As described above, the operating system assigns a unique process identification number (proc_id) to each spawned process. Each process has a name, and each process knows the names of other processes with which it needs to communicate. The operating system keeps a list of process names and the assigned process identification numbers. Processes send messages to other processes using the assigned process identification numbers without regard to what board is executing each process (i.e., process location).

Application Programming Interfaces (APIs) define the format and type of information included in the messages.

The modular software architecture configuration model requires a single software process to support multiple configurable objects. For example, as described above, an ATM application may support configurations requiring multiple ATM interfaces and thousands of permanent virtual connections per ATM interface. The number of processes and configurable objects in a modular software architecture can quickly grow especially in a distributed processing system. If the operating system assigns a new process for each configurable object, the operating system's capabilities may be quickly exceeded.

For example, the operating system may be unable to assign a process for each ATM interface, each service endpoint, each permanent virtual circuit, etc.. In some instances, the process identification numbering scheme itself may not be large enough. Where protected memory is supported, the system may have insufficient memory to assign each process and configurable object a separate memory block. In addition, supporting a large number of independent processes may reduce the operating system's efficiency and slow the operation of the entire computer system.

One alternative is to assign a unique process identification number to only certain high level processes. Referring to Fig. 16a, for example, process identification numbers may only be assigned to each ATM process (e.g., ATMs 240, 241) and not to each ATM interface (e.g., ATM IFs 242-247) and process identification numbers may only be assigned to each port device driver (e.g., device drivers 248, 250, 252) and not to each service endpoint (e.g., SE 253-261). A disadvantage to this approach is that objects within one high level process will likely need to communicate with objects within other high level processes. For example, ATM interface 242 within ATM 240 may need to communicate with SE 253 within device driver 248. ATM IF 242 needs to know if SE 253 is active and perhaps certain other information about SE 253. Since SE 253 was not assigned a process identification number, however, neither ATM 240 nor ATM IF 242 knows if it exists. Similarly, ATM IF 242 knows it needs to communicate with SE 253 but does not know that device driver 248 controls SE 253.

One possible solution is to hard code the name of device driver 248 into ATM 240. ATM 240 then knows it must communicate with device driver 248 to learn about the existence of any service endpoints within device driver 248 that may be needed by ATM IF 242, 243 or 244. Unfortunately, this can lead to scalability issues. For instance, each instantiation of ATM (e.g., ATM 240, 241) needs to know the name of all device drivers (e.g., device drivers 248, 250, 252) and must query each device driver to locate each needed service endpoint. An ATM query to a device driver that does not include a necessary service endpoint is a waste of time and resources. In addition, each high level process must periodically poll other high level processes to determine whether objects within them are still active (i.e., not terminated) and whether new objects have been started. If the object status has not changed between polls, then the poll wasted resources. If the status did change, then communications have been stalled for the length of time between polls. In addition, if a new device driver is added (e.g., device driver 262), then ATM 240 and 241 cannot communicate with it or any of the service endpoints within it until they have been upgraded to include the new device driver's name.

Processes that use this API must use the predefined message format. If a process is upgraded, it will be forced to use the same message format or change the API / message format which would require that all processes that use this API also be similarly upgraded to use the new API. Instead, the message format can be made more flexible by passing information through the name server or BOM. For example, instead of having the value field 274 be a fixed number of bits, when an application registers a name and process identification number it may also register the number of bits it plans on using for the value field (or any other field). Perhaps a zero indicates a value field of 32 bits and a one indicates a value field of 64 bits. Thus, both processes know the message format but some flexibility has been added.

In addition to adding flexibility to the size of fields in a message format, flexibility may be added to the overall message format including the type of fields included in the message. When a process registers its name and process identification number, it may also register a version number indicating which API version should be used by other processes wishing to communicate with it. For example, device driver 250 (Fig. 16b) may register SE 258 with NS 264 and provide the name of SE 258, device driver 250's process identification number and a version number one, and device driver 252 may register SE 261 with NS 264 and provide the name of SE 261, device driver 252's process identification number and a version number (e.g., version number two). If ATM 240 has subscribed for either SE 258 or SE 261, then NS 264 notifies ATM 240 that SE 258 and SE 261 exist and provides the process identification numbers and version numbers. The version number tells ATM 240 what message format and information SE 258 and SE 261 expect. The different message formats for each version may be hard coded into ATM 240 or ATM 240 may access system memory or the configuration database for the message formats corresponding to service endpoint version one and version two. As a result, the same application may communicate with different versions of the same configurable object using a different API.

This also allows an application, for example, ATM, to be upgraded to support new configurable objects, for example, new ATM interfaces, while still being backward

changes are possible, and the modular software architecture allows them to be made while the computer system is running (hot changes). Configuration changes may be automatically copied to persistent storage as they are made so that if the computer system is shut down and rebooted, the memory and configuration database will reflect the last known state of the hardware.

To make a configuration change, the user informs the NMS (e.g., NMS client 850a, Fig. 2a) of the particular change, and similar to the process for initial configuration, the NMS (e.g., NMS server 851a, Fig. 2a) changes the appropriate tables in the configuration database (copied to the NMS database) to implement the change.

Referring to Fig. 17, in one example of a configuration change, the user notifies the NMS that an additional path will be carried by SONET fiber 70c connected to port 44c. A new service endpoint (SE) 164 and a new ATM interface 166 are needed to handle the new path. The NMS adds a new record (row 168, Fig. 10) to service endpoint table (SET) 76 to include service endpoint 10 corresponding to port physical identification number (PID) 1502 (port 44c). The NMS also adds a new record (row 170, Fig. 13) to ATM instance table 114 to include ATM interface (IF) 12 corresponding to ATM group 3 and SE 10. Configuration database 42 may automatically copy the changes made to SET 76 and ATM instance table 114 to persistent storage 21 such that if the computer system is shut down and rebooted, the changes to the configuration database will be maintained.

Configuration database 42 also notifies (through the active query process) SEM 96c that a new service endpoint (SE 10) was added to the SET corresponding to its port (PID 1502), and configuration database 42 also notifies ATM instantiation 112 that a new ATM interface (ATM-IF 166) was added to the ATM interface table corresponding to ATM group 3. ATM 112 establishes ATM interface 166 and SEM 96c notifies port driver 142 that it has been assigned SE10. A communication link is established through NS 220b. Device driver 142 generates a service endpoint name using the assigned SE number and publishes this name and its process identification number with NS 220b. ATM interface 166 generates the same service endpoint name and subscribes to NS 220b

for that service endpoint name. NS 220b provides ATM interface 166 with the process identification assigned to DD 142 allowing ATM interface 166 to communicate with device driver 142.

Certain board changes to computer system 10 are also configuration changes. After power-up and configuration, a user may plug another board into an empty computer system slot or remove an enabled board and replace it with a different board. In the case where applications and drivers for a line card added to computer system 10 are already loaded, the configuration change is similar to initial configuration. The additional line card may be identical to an already enabled line card, for example, line card 16a or if the additional line card requires different drivers (for different components) or different applications (e.g., IP), the different drivers and applications are already loaded because computer system 10 expects such cards to be inserted.

Referring to Fig. 18, while computer system 10 is running, when another line card 168 is inserted, master MCD 38 detects the insertion and communicates with a diagnostic program 170 being executed by the line card's processor 172 to learn the card's type and version number. MCD 38 uses the information it retrieves to update card table 47 and port table 49. MCD 38 then searches physical module description (PMD) file 48 in memory 40 for a record that matches the retrieved card type and version number and retrieves the name of the mission kernel image executable file (MKI.exe) that needs to be loaded on line card 168. Once determined, master MCD 38 passes the name of the MKI executable file to master SRM 36. SRM 36 downloads MKI executable file 174 from persistent storage 21 and passes it to a slave SRM 176 running on line card 168. The slave SRM executes the received MKI executable file.

Referring to Fig. 19, slave MCD 178 then searches PMD file 48 in memory 40 on central processor 12 for a match with its line card's type and version number to find the names of all the device driver executable files associated needed by its line card. Slave MCD 178 provides these names to slave SRM 176 which then downloads and executes the device driver executable files (DD.exe) 180 from memory 40.

When master MCD 38 updates card table 47, configuration database 42 updated NMS database 61 which sends NMS 60 (e.g., NMS Server 851a, Fig. 2a) a notification of the change including card type and version number, the slot number into which the card was inserted and the physical identification (PID) assigned to the card by the master MCD. The NMS is updated, assigns an LID and updates the logical to physical table and notifies the user of the new hardware. The user then tells the NMS how to configure the new hardware, and the NMS implements the configuration change as described above for initial configuration.

Logical Model Change:

Where applications and device drivers for a new line card are not already loaded and where changes or upgrades to already loaded applications and device drivers are needed, logical model 280 (Figs. 2a-3e) must be changed and new view ids and APIs, NMS JAVA interface files, persistent layer metadata files and new DDL files must be re-generated. Software model 286 is changed to include models of the new or upgraded software, and hardware model 284 is changed to include models of any new hardware. New logical model 280' is then used by code generation system 336 to re-generate view ids and APIs for each application, including any new applications, for example, ATM version two 360, or device drivers, for example, device driver 362, and to re-generate DDL files 344' and 348' including new SQL commands and data relevant to the new hardware and/or software. The new logical model is also used to generate new NMS JAVA interface files 347' and new persistent layer metadata files 349'. Each application, including any new applications or drivers, is then pulled into the build process and links in a corresponding view id and API. The new applications and/or device drivers, NMS JAVA interface files, new persistent layer metadata files and the new DDL files as well as any new hardware are then sent to the user of computer system 10.

New and upgraded applications and device drivers are being used by way of an example, and it should be understood that other processes, for example, modular system services

The NMS then writes a new record into SMS table 192 indicating the scope of the configuration update. The scope of an upgrade may be indicated in a variety of ways. In one embodiment, the SMS table includes a field for the name of the application to be changed and other fields indicating the changes to be made. In another embodiment, the SMS table includes a revision number field 194 (Fig. 21) through which the NMS can indicate the scope of the change. Referring to Fig. 21, the right most position in the revision number may indicate, for example, the simplest configuration update (e.g., a bug fix), in this case, termed a "service update level" 196. Any software revisions that differ by only the service update level can be directly applied without making changes in the configuration database or API changes between the new and current revision. The next position may indicate a slightly more complex update, in this case, termed a "subsystem compatibility level" 198. These changes include changes to the configuration database and/or an API. The next position may indicate a "minor revision level" 200 update indicating more comprehensive changes in both the configuration database and one or more APIs. The last position may indicate a "major revision level" 202 update indicative of wholesale changes in multiple areas and may require a reboot of the computer system to implement. For a major revision level change, the NMS will download a complete image including a kernel image.

During initial configuration, the SMS establishes an active query on SMS table 192. Consequently, when the NMS changes the SMS table, the configuration database sends a notification to master SMS 184 including the change. In some instances, the change to an application may require changes to configuration database 42. The SMS determines the need for configuration conversion based on the scope of the release or update. If the configuration database needs to be changed, then the software, for example, ATM version two 360, provided by the user and downloaded by the NMS also includes a configuration control executable file, for example, ATMv2_cnfig_cntrl.exe 191, and the name of this file will be in the SMS table record. The master SMS then directs slave SRM 37a on central processor 12 to execute the configuration control file which uses DDL file 344' to upgrade old configuration database 42 into new configuration database

42' by creating new tables, for example, ATM group table 108' and ATM interface table 114'.

Existing processes using their view ids and APIs to access new configuration database 42' in the same manner as they accessed old configuration database 42. However, when new processes (e.g., ATM version two 360 and device driver 362) access new configuration database 42', their view ids and APIs allow them to access new tables and data within new configuration database 42'.

The master SMS also reads ATM group table 108' to determine that instances of ATM are being executed on line cards 16a-16n. In order to upgrade a distributed application, in this instance, ATM, the Master SMS will use a lock step procedure. Master SMS 184 tells each slave SMS 186b-186n to stall the current versions of ATM. When each slave responds, Master SMS 184 then tells slave SMSs 186b-186n to download and execute ATMv2_cntrl.exe 190 from memory 40. Upon instructions from the slave SMSs, slave SRMs 37b-37n download and execute copies of ATMv2_cntrl.exe 204a-204n. The slave SMSs also pass data to the ATMv2cntrl.exe file through the SRM. The data instructs the control shim to start in upgrade mode and passes required configuration information. The upgraded ATMv2 controllers 204a-204n then use ATM group table 108' and ATM interface table 114' as described above to implement ATMv2 206a-206n on each of the line cards. In this example, each ATM controller is shown implementing one instance of ATM on each line card, but as explained below, the ATM controller may implement multiple instances of ATM on each line card.

As part of the upgrade mode, the updated versions of ATMv2 206a-206n retrieve active state from the current versions of ATM 188a-188n. The retrieval of active state can be accomplished in the same manner that a redundant or backup instantiation of ATM retrieves active state from the primary instantiation of ATM. When the upgraded instances of ATMv2 are executing and updated with active state, the ATMv2 controllers notify the slave SMSs 186b-186n on their board and each slave SMS 186b-186n notifies master SMS 184. When all boards have notified the master SMS, the master SMS tells

the slave SMSs to switchover to ATMv2 206a-206n. The slave SMSs tell the slave SRMs running on their board, and the slave SRMs transition the new ATMv2 processes to the primary role. This is termed “lock step upgrade” because each of the line cards is switched over to the new ATMv2 processes simultaneously.

There may be upgrades that require changes to multiple applications and to the APIs for those applications. For example, a new feature may be added to ATM that also requires additional functionality to be added to the Multi-Protocol Label Switching (MPLS) application. The additionally functionality may change the peer-to-peer API for ATM, the peer-to-peer API for MPLS and the API between ATM and MPLS. In this scenario, the upgrade operation must avoid allowing the “new” version of ATM to communicate with itself or the “old” version of MPLS and vice versa. The master SMS will use the release number scheme to determine the requirements for the individual upgrade. For example, the upgrade may be from release 1.0.0.0 to 1.0.1.3 where the release differs by the subsystem compatibility level. The SMS implements the upgrade in a lock step fashion. All instances of ATM and MPLS are upgraded first. The slave SMS on each line card then directs the slave SRM on its board to terminate all “old” instances of ATM and MPLS and switchover to the new instances of MPLS and ATM. The simultaneous switchover to new versions of both MPLS and ATM eliminate any API compatibility errors.

Referring to Fig. 22, instead of directly upgrading configuration database 42 on central processor 12, a backup configuration database 420 on a backup central processor 13 may be upgraded first. As described above, computer system 10 includes central processor 12. Computer system 10 may also include a redundant or backup central processor 13 that mirrors or replicates the active state of central processor 12. Backup central processor 13 is generally in stand-by mode unless central processor 12 fails at which point a fail-over to backup central processor 13 is initiated to allow the backup central processor to be substituted for central processor 12. In addition to failures, backup central processor 13 may be used for software and hardware upgrades that require changes to the configuration database. Through backup central processor 13, upgrades

can be made to backup configuration database 420 instead of to configuration database 42.

The upgrade is begun as discussed above with the NMS downloading ATM version two 360 – including ATMv2.exe 189, ATMv2_cntrl.exe and ATMv2_cnfg_cntrl.exe – and DDL file 344' to memory on central processor 12. Simultaneously, because central processor 13 is in backup mode, the application and DDL file are also copied to memory on central processor 13. The NMS also creates a software load record in SMS table 192, 192' indicating the upgrade. In this embodiment, when the SMS determines that the scope of the upgrade requires an upgrade to the configuration database, the master SMS instructs slave SMS 186e on central processor 13 to perform the upgrade. Slave SMS 186e works with slave SRM 37e to cause backup processor 13 to change from backup mode to upgrade mode.

In upgrade mode, backup processor 13 stops replicating the active state of central processor 12. Any changes made to new configuration database 420 are copied to new NMS database 61'. Slave SMS 186e then directs slave SRM 37e to execute the configuration control file which uses DDL file 344' to upgrade configuration database 420.

Once configuration database 420 is upgraded, a fail-over or switch-over from central processor 12 to backup central processor 13 is initiated. Central processor 13 then begins acting as the primary central processor and applications running on central processor 13 and other boards throughout computer system 10 begin using upgraded configuration database 420.

Central processor 12 may not become the backup central processor right away. Instead, central processor 12 with its older copy of configuration database 42 stays dormant in case an automatic downgrade is necessary (described below). If the upgrade goes smoothly and is committed (described below), then central processor 12 will begin

operating in backup mode and replace old configuration database 42 with new configuration database 420.

Device Driver Upgrade:

Device driver software may also be upgraded and the implementation of device driver upgrades is similar to the implementation of application upgrades. The user informs the NMS of the device driver change and provides a copy of the new software (e.g., DD[^].exe 362, Figs. 20 and 23). The NMS downloads the new device driver to memory 40 on central processor 12, and the NMS writes a new record in SMS table 192 indicating the device driver upgrade. Configuration database 42 sends a notification to master SMS 184 including the name of the driver to be upgraded. To determine where the original device driver is currently running in computer system 10, the master SMS searches PMD file 48 for a match of the device driver name (existing device driver, not upgraded device driver) to learn with which module type and version number the device driver is associated. The device driver may be running on one or more boards in computer system 10. As described above, the PMD file corresponds the module type and version number of a board with the mission kernel image for that board as well as the device drivers for that board. The SMS then searches card table 47 for a match with the module type and version number found in the PMD file. Card table 47 includes records corresponding module type and version number with the physical identification (PID) and slot number of that board. The master SMS now knows the board or boards within computer system 10 on which to load the upgraded device driver. If the device driver is for a particular port, then the SMS must also search the port table to learn the PID for that port.

The master SMS notifies each slave SMS running on boards to be upgraded of the name of the device driver executable file to download and execute. In the example, master SMS 184 sends slave SMS 186f the name of the upgraded device driver (DD[^].exe 362) to download. Slave SMS 186f tells slave SRM to download and execute DD[^].exe 362 in upgrade mode. Once downloaded, DD[^].exe 363 (copy of DD[^].exe 362) gathers active state information from the currently running DD.exe 212 in a similar fashion as a redundant or backup device driver would gather active state. DD[^].exe 362 then notifies

slave SRM 37f that active state has been gathered, and slave SRM 37f stops the current DD.exe 212 process and transitions the upgraded DD^.exe 362 process to the primary role.

Automatic Downgrade:

Often, implementation of an upgrade, can cause unexpected errors in the upgraded software, in other applications or in hardware. As described above, a new configuration database 42' (Fig. 20) is generated and changes to the new configuration database are made in new tables (e.g., ATM interface table 114' and ATM group table 108', Fig. 20) and new executable files (e.g., ATMv2.exe 189, ATMv2_cntrl.exe 190 and ATMv2_cnfg_cntrl.exe 191) are downloaded to memory 40. Importantly, the old configuration database records and the original application files are not deleted or altered. In the embodiment where changes are made directly to configuration database 42 on central processor 12, they are made only in non-persistent memory until committed (described below). In the embodiment where changes are made to backup configuration database 420 on backup central processor 13, original configuration database 42 remains unchanged.

Because the operating system provides a protected memory model that assigns different process blocks to different processes, including upgraded applications, the original applications will not share memory space with the upgraded applications and, therefore, cannot corrupt or change the memory used by the original application. Similarly, memory 40 is capable of simultaneously maintaining the original and upgraded versions of the configuration database records and executable files as well as the original and upgraded versions of the applications (e.g., ATM 188a-188n). As a result, the SMS is capable of an automatic downgrade on the detection of an error. To allow for automatic downgrade, the SRMs pass error information to the SMS. The SMS may cause the system to revert to the old configuration and application (i.e., automatic downgrade) on any error or only for particular errors.

ATMv2 interface table 114' that the upgrade is for only one instantiation (e.g., one port) on line card 16a. Consequently, a failure is likely to only disrupt service on that one port, and again, the SMS can further minimize the disruption by automatically downgrading the configuration of that port on the detection of an error. If no error is detected during the evaluation mode, then the upgrade can be implemented over the entire computer system.

Upgrade Commitment:

Upgrades are made permanent by saving the new application software and new configuration database and DDL file in persistent storage and removing the old configuration data from memory 40 as well as persistent storage. As mentioned above, changes may be automatically saved in persistent storage as they are made in non-persistent memory (no automatic downgrade), or the user may choose to automatically commit an upgrade after a successful time interval lapses (evaluation mode). The time interval from upgrade to commitment may be significant. During this time, configuration changes may be made to the system. Since these changes are typically made in non-persistent memory, they will be lost if the system is rebooted prior to upgrade commitment. Instead, to maintain the changes, the user may request that certain configuration changes made prior to upgrade commitment be copied into the old configuration database in persistent memory. Alternatively, the user may choose to manually commit the upgrade at his or her leisure. In the manual mode, the user would ask the NMS to commit the upgrade and the NMS would inform the master SMS, for example, through a record in the SMS table.

Independent Process Failure and Restart:

Depending upon the fault policy managed by the slave SRMs on each board, the failure of an application or device driver may not immediately cause an automatic downgrade during an upgrade process. Similarly, the failure of an application or device driver during normal operation may not immediately cause the fail over to a backup or redundant board. Instead, the slave SRM running on the board may simply restart the failing

Vertical fault isolation allows processes to be deployed in a fashion supportive of the underlying hardware architecture and allows processes associated with particular hardware (e.g., a port) to be isolated from processes associated with other hardware (e.g., other ports) on the same or a different line card. Any single hardware or software failure will affect only those customers serviced by the same vertical stack. Vertical fault isolation provides a fine grain of fault isolation and containment. In addition, recovery time is reduced to only the time required to re-start a particular application or driver instead of the time required to re-start all the processes associated with a line card or the entire system.

Fault / Event Detection:

Traditionally, fault detection and monitoring does not receive a great deal of attention from network equipment designers. Hardware components are subjected to a suite of diagnostic tests when the system powers up. After that, the only way to detect a hardware failure is to watch for a red light on a board or wait for a software component to fail when it attempts to use the faulty hardware. Software monitoring is also reactive. When a program fails, the operating system usually detects the failure and records minimal debug information.

Current methods provide only sporadic coverage for a narrow set of hard faults. Many subtler failures and events often go undetected. For example, hardware components sometimes suffer a minor deterioration in functionality, and changing network conditions stress the software in ways that were never expected by the designers. At times, the software may be equipped with the appropriate instrumentation to detect these problems before they become hard failures, but even then, network operators are responsible for manually detecting and repairing the conditions.

Systems with high availability goals must adopt a more proactive approach to fault and event monitoring. In order to provide comprehensive fault and event detection, different hierarchical levels of fault/event management software are provided that intelligently

monitor hardware and software and proactively take action in accordance with a defined fault policy. A fault policy based on hierarchical scopes ensures that for each particular type of failure the most appropriate action is taken. This is important because over-reacting to a failure, for example, re-booting an entire computer system or re-starting an entire line card, may severely and unnecessarily impact service to customers not affected by the failure, and under-reacting to failures, for example, restarting only one process, may not completely resolve the fault and lead to additional, larger failures. Monitoring and proactively responding to events may also allow the computer system and network operators to address issues before they become failures. For example, additional memory may be assigned to programs or added to the computer system before a lack of memory causes a failure.

Hierarchical Scopes and Escalation:

Referring to Fig. 26, in one embodiment, master SRM 36 serves as the top hierarchical level fault/event manager, each slave SRM 37a-37n serves as the next hierarchical level fault/event manager, and software applications resident on each board, for example, ATM 110-113 and device drivers 43a-43d on line card 16a include sub-processes that serve as the lowest hierarchical level fault/event managers (i.e., local resiliency managers, LRM). Master SRM 36 downloads default fault policy (DFP) files (metadata) 430a-430n from persistent storage to memory 40. Master SRM 36 reads a master default fault policy file (e.g., DFP 430a) to understand its fault policy, and each slave SRM 37a-37n downloads a default fault policy file (e.g., DFP 430b-430n) corresponding to the board on which the slave SRM is running. Each slave SRM then passes to each LRM a fault policy specific to each local process.

A master logging entity 431 also runs on central processor 12 and slave logging entities 433a-433n run on each board. Notifications of failures and other events are sent by the master SRM, slave SRMs and LRMs to their local logging entity which then notifies the master logging entity. The master logging entity enters the event in a master event log file 435. Each local logging entity may also log local events in a local event log file 435a-435n.

In addition, a fault policy table 429 may be created in configuration database 42 by the NMS when the user wishes to over-ride some or all of the default fault policy (see configurable fault policy below), and the master and slave SRMs are notified of the fault policies through the active query process.

Referring to Fig. 27, as one example, ATM application 110 includes many sub-processes including, for example, an LRM program 436, a Private Network-to-Network Interface (PNNI) program 437, an Interim Link Management Interface (ILMI) program 438, a Service Specific Connection Oriented Protocol (SSCOP) program 439, and an ATM signaling (SIG) program 440. ATM application 110 may include many other sub-programs only a few have been shown for convenience. Each sub-process may also include sub-processes, for example, ILMI sub-processes 438a-438n. In general, the upper level application (e.g., ATM 110) is assigned a process memory block that is shared by all its sub-processes.

If, for example, SSCOP 439 detects a fault, it notifies LRM 436. LRM 436 passes the fault to local slave SRM 37b, which catalogs the fault in the ATM application's fault history and sends a notice to local slave logging entity 433b. The slave logging entity sends a notice to master logging entity 431, which may log the event in master log event file 435. The local logging entity may also log the failure in local event log 435a. LRM 436 also determines, based on the type of failure, whether it can fully resolve the error and do so without affecting other processes outside its scope, for example, ATM 111-113, device drivers 43a-43d and their sub-processes and processes running on other boards. If yes, then the LRM takes corrective action in accordance with its fault policy. Corrective action may include restarting SSCOP 439 or resetting it to a known state.

Since all sub-processes within an application, including the LRM sub-process, share the same memory space, it may be insufficient to restart or reset a failing sub-process (e.g., SSCOP 439). Hence, for most failures, the fault policy will cause the LRM to escalate the failure to the local slave SRM. In addition, many failures will not be presented to the

LRM but will, instead, be presented directly to the local slave SRM. These failures are likely to have been detected by either processor exceptions, OS errors or low-level system service errors. Instead of failures, however, the sub-processes may notify the LRM of events that may require action. For example, the LRM may be notified that the PNNI message queue is growing quickly. The LRM's fault policy may direct it to request more memory from the operating system. The LRM will also pass the event to the local slave SRM as a non-fatal fault. The local slave SRM will catalog the event and log it with the local logging entity, which may also log it with the master logging entity. The local slave SRM may take more severe action to recover from an excessive number of these non-fatal faults that result in memory requests.

If the event or fault (or the actions required to handle either) will affect processes outside the LRM's scope, then the LRM notifies slave SRM 37b of the event or failure. In addition, if the LRM detects and logs the same failure or event multiple times and in excess of a predetermined threshold set within the fault policy, the LRM may escalate the failure or event to the next hierarchical scope by notifying slave SRM 37b. Alternatively or in addition, the slave SRM may use the fault history for the application instance to determine when a threshold is exceeded and automatically execute its fault policy.

When slave SRM 37b detects or is notified of a failure or event, it notifies slave logging entity 435b. The slave logging entity notifies master logging entity 431, which may log the failure or event in master event log 435, and the slave logging entity may also log the failure or event in local event log 435b. Slave SRM 37b also determines, based on the type of failure or event, whether it can handle the error without affecting other processes outside its scope, for example, processes running on other boards. If yes, then slave SRM 37b takes corrective action in accordance with its fault policy and logs the fault. Corrective action may include re-starting one or more applications on line card 16a.

If the fault or recovery actions will affect processes outside the slave SRM's scope, then the slave SRM notifies master SRM 36. In addition, if the slave SRM has detected and logged the same failure multiple times and in excess of a predetermined threshold, then

the slave SRM may escalate the failure to the next hierarchical scope by notifying master SRM 36 of the failure. Alternatively, the master SRM may use its fault history for a particular line card to determine when a threshold is exceeded and automatically execute its fault policy.

When master SRM 36 detects or receives notice of a failure or event, it notifies slave logging entity 433a, which notifies master logging entity 431. The master logging entity 431 may log the failure or event in master log file 435 and the slave logging entity may log the failure or event in local event log 435a. Master SRM 36 also determines the appropriate corrective action based on the type of failure or event and its fault policy. Corrective action may require failing-over one or more line cards 16a-16n or other boards, including central processor 12, to redundant backup boards or, where backup boards are not available, simply shutting particular boards down. Some failures may require the master SRM to re-boot the entire computer system.

An example of a common error is a memory access error. As described above, when the slave SRM starts a newinstance of an application, it requests a protected memory block from the local operating system. The local operating systems assign each instance of an application one block of local memory and then program the local memory management unit (MMU) hardware with which processes have access (read and/or write) to each block of memory. An MMU detects a memory access error when a process attempts to access a memory block not assigned to that process. This type of error may result when the process generates an invalid memory pointer. The MMU prevents the failing process from corrupting memory blocks used by other processes (i.e., protected memory model) and sends a hardware exception to the local processor. A local operating system fault handler detects the hardware exception and determines which process attempted the invalid memory access. The fault handler then notifies the local slave SRM of the hardware exception and the process that caused it. The slave SRM determines the application instance within which the fault occurred and then goes through the process described above to determine whether to take corrective action, such as restarting the application, or escalate the fault to the master SRM.

As another example, a device driver, for example, device driver 43a may determine that the hardware associated with its port, for example, port 44a, is in a bad state. Since the failure may require the hardware to be swapped out or failed-over to redundant hardware or the device driver itself to be re-started, the device driver notifies slave SRM 37b. The slave SRM then goes through the process described above to determine whether to take corrective action or escalate the fault to the master SRM.

As a third example, if a particular application instance repeatedly experiences the same software error but other similar application instances running on different ports do not experience the same error, the slave SRM may determine that it is likely a hardware error. The slave SRM would then notify the master SRM which may initiate a fail-over to a backup board or, if no backup board exists, simply shut down that board or only the failing port on that board. Similarly, if the master SRM receives failure reports from multiple boards indicating Ethernet failures, the master SRM may determine that the Ethernet hardware is the problem and initiate a fail-over to backup Ethernet hardware.

Consequently, the failure type and the failure policy determine at what scope recovery action will be taken. The higher the scope of the recovery action, the larger the temporary loss of services. Speed of recovery is one of the primary considerations when establishing a fault policy. Restarting a single software process is much faster than switching over an entire board to a redundant board or re-booting the entire computer system. When a single process is restarted, only a fraction of a card's services are affected. Allowing failures to be handled at appropriate hierarchical levels avoids unnecessary recovery actions while ensuring that sufficient recovery actions are taken, both of which minimize service disruption to customers.

Hierarchical Descriptors:

Hierarchical descriptors may be used to provide information specific to each failure or event. The hierarchical descriptors provide granularity with which to report faults, take action based on fault history and apply fault recovery policies. The descriptors can be

stored in master event log file 435 or local event log files 435a-435n through which faults and events may be tracked and displayed to the user and allow for fault detection at a fine granular level and proactive response to events. In addition, the descriptors can be matched with descriptors in the fault policy to determine the recovery action to be taken.

Referring to Fig. 28, in one embodiment, a descriptor 441 includes a top hierarchical class field 442, a next hierarchical level sub-class field 444, a lower hierarchical level type field 446 and a lowest level instance field 448. The class field indicates whether the failure or event is related (or suspected to relate) to hardware or software. The subclass field categorizes events and failures into particular hardware or software groups. For example, under the hardware class, subclass indications may include whether the fault or event is related to memory, Ethernet, switch fabric or network data transfer hardware. Under the software class, subclass indications may include whether the fault or event is a system fault, an exception or related to a specific application, for example, ATM.

The type field more specifically defines the subclass failure or event. For example, if a hardware class, Ethernet subclass failure has occurred, the type field may indicate a more specific type of Ethernet failure, for instance, a cyclic redundancy check (CRC) error or a runt packet error. Similarly, if a software class, ATM failure or event has occurred, the type field may indicate a more specific type of ATM failure or event, for instance, a private network-to-network interface (PNNI) error or a growing message queue event. The instance field identifies the actual hardware or software that failed or generated the event. For example, with regard to a hardware class, Ethernet subclass, CRC type failure, the instance indicates the actual Ethernet port that experienced the failure. Similarly, with regard to a software class, ATM subclass, PNNI type, the instance indicates the actual PNNI sub-program that experienced the failure or generated the event.

When a fault or event occurs, the hierarchical scope that first detects the failure or event creates a descriptor by filling in the fields described above. In some cases, however, the Instance field is not applicable. The descriptor is sent to the local logging entity, which may log it in the local event log file before notifying the master logging entity, which

may log it in the master event log file 435. The descriptor may also be sent to the local slave SRM, which tracks fault history based on the descriptor contents per application instance. If the fault or event is escalated, then the descriptor is passed to the next higher hierarchical scope.

When slave SRM 37b receives the fault / event notification and the descriptor, it compares it to descriptors in the fault policy for the particular scope in which the fault occurred looking for a match or a best case match which will indicate the recovery procedure to follow. Fault descriptors within the fault policy can either be complete descriptors or have wildcards in one or more fields. Since the descriptors are hierarchical from left to right, wildcards in descriptor fields only make sense from right to left. The fewer the fields with wildcards, the more specific the descriptor. For example, a particular fault policy may apply to all software faults and would, therefore, include a fault descriptor having the class field set to "software" and the remaining fields - subclass, type, and instance - set to wildcard or "match all." The slave SRM searches the fault policy for the best match (i.e., the most fields matched) with the descriptor to determine the recovery action to be taken.

Configurable Fault Policy:

In actual use, a computer system is likely to encounter scenarios that differ from those in which the system was designed and tested. Consequently, it is nearly impossible to determine all the ways in which a computer system might fail, and in the face of an unexpected error, the default fault policy that was shipped with the computer system may cause the hierarchical scope (master SRM, slave SRM or LRM) to under-react or over-react. Even for expected errors, after a computer system ships, certain recovery actions in the default fault policy may be determined to be over aggressive or too lenient. Similar issues may arise as new software and hardware is released and/or upgraded.

A configurable fault policy allows the default fault policy to be modified to address behavior specific to a particular upgrade or release or to address behavior that was learned after the implementation was released. In addition, a configurable fault policy

allows users to perform manual overrides to suit their specific requirements and to tailor their policies based on the individual failure scenarios that they are experiencing.

The modification may cause the hierarchical scope to react more or less aggressively to particular known faults or events, and the modification may add recovery actions to handle newly learned faults or events. The modification may also provide a temporary patch while a software or hardware upgrade is developed to fix a particular error.

If an application runs out of memory space, it notifies the operating system and asks for more memory. For certain applications, this is standard operating procedure. As an example, an ATM application may have set up a large number of virtual circuits and to continue setting up more, additional memory is needed. For other applications, a request for more memory indicates a memory leak error. The fault policy may require that the application be re-started causing some service disruption. It may be that re-starting the application eventually leads to the same error due to a bug in the software. In this instance, while a software upgrade to fix the bug is developed, a temporary patch to the fault policy may be necessary to allow the memory leak to continue and prevent repeated application re-starts that may escalate to line card re-start or fail-over and eventually to a re-boot of the entire computer system. A temporary patch to the default fault policy may simply allow the hierarchical scope, for example, the local resiliency manager or the slave SRM, to assign additional memory to the application. Of course, an eventual re-start of the application is likely to be required if the application's leak consumes too much memory.

A temporary patch may also be needed while a hardware upgrade or fix is developed for a particular hardware fault. For instance, under the default fault policy, when a particular hardware fault occurs, the recovery policy may be to fail-over to a backup board. If the backup board includes the same hardware with the same hardware bug, for example, a particular semiconductor chip, then the same error will occur on the backup board. To prevent a repetitive fail-over while a hardware fix is developed, the temporary patch to the default fault policy may be to restart the device driver associated with the particular hardware instead of failing-over to the backup board.

In addition to the above needs, a configurable fault policy also allows purchasers of computer system 10 (e.g., network service providers) to define their own policies. For example, a network service provider may have a high priority customer on a particular port and may want all errors and events (even minor ones) to be reported to the NMS and displayed to the network manager. Watching all errors and events might give the network manager early notice of growing resource consumption and the need to plan to dedicate additional resources to this customer.

As another example, a user of computer system 10 may want to be notified when any process requests more memory. This may give the user early notice of the need to add more memory to their system or to move some customers to different line cards.

Referring again to Fig. 26, to change the default fault policy as defined by default fault policy (DFP) files 430a-430n, a configuration fault policy file 429 is created by the NMS in the configuration database. An active query notification is sent by the configuration database to the master SRM indicating the changes to the default fault policy. The master SRM notifies any slave SRMs of any changes to the default fault policies specific to the boards on which they are executing, and the slave SRMs notify any LRMs of any changes to the default fault policies specific to their process. Going forward, the default fault policies - as modified by the configuration fault policy - are used to detect, track and respond to events or failures.

Alternatively, active queries may be established with the configuration database for configuration fault policies specific to each board type such that the slave SRMs are notified directly of changes to their default fault policies.

A fault policy (whether default or configured) is specific to a particular scope and descriptor and indicates a particular recovery action to take. As one example, a temporary patch may be required to handle hardware faults specific to a known bug in an integrated circuit chip. The configured fault policy, therefore, may indicate a scope of all

line cards, if the component is on all line cards, or only a specific type of line card that includes that component. The configured fault policy may also indicate that it is to be applied to all hardware faults with that scope, for example, the class will indicate hardware (HW) and all other fields will include wildcards (e.g., HW.*.*.*). Instead, the configured fault policy may only indicate a particular type of hardware failure, for example, CRC errors on transmitted Ethernet packets (e.g., HW.Ethernet.TxCRC.*).

Redundancy:

As previously mentioned, a major concern for service providers is network downtime. In pursuit of “five 9’s availability” or 99.999% network up time, service providers must minimize network outages due to equipment (i.e., hardware) and all too common software failures. Developers of computer systems often use redundancy measures to minimize downtime and enhance system resiliency. Redundant designs rely on alternate or backup resources to overcome hardware and/or software faults. Ideally, the redundancy architecture allows the computer system to continue operating in the face of a fault with minimal service disruption, for example, in a manner transparent to the service provider’s customer.

Generally, redundancy designs come in two forms: 1:1 and 1:N. In a so-called “1:1 redundancy” design, a backup element exists for every active or primary element (i.e., hardware backup). In the event that a fault affects a primary element, a corresponding backup element is substituted for the primary element. If the backup element has not been in a “hot” state (i.e., software backup), then the backup element must be booted, configured to operate as a substitute for the failing element, and also provided with the “active state” of the failing element to allow the backup element to take over where the failed primary element left off. The time required to bring the software on the backup element to an “active state” is referred to as synchronization time. A long synchronization time can significantly disrupt system service, and in the case of a computer network device, if synchronization is not done quickly enough, then hundreds or thousands of network connections may be lost which directly impacts the service provider’s availability statistics and angers network customers.

To minimize synchronization time, many 1:1 redundancy schemes support hot backup of software, which means that the software on the backup elements mirror the software on the primary elements at some level. The “hotter” the backup element – that is, the closer the backup mirrors the primary – the faster a failed primary can be switched over or failed over to the backup. The “hottest” backup element is one that runs hardware and software simultaneously with a primary element conducting all operations in parallel with the primary element. This is referred to as a “1+1 redundancy” design and provides the fastest synchronization.

Significant costs are associated with 1:1 and 1+1 redundancy. For example, additional hardware costs may include duplicate memory components and printed circuit boards including all the components on those boards. The additional hardware may also require a larger supporting chassis. Space is often limited, especially in the case of network service providers who may maintain hundreds of network devices. Although 1:1 redundancy improves system reliability, it decreases service density and decreases the mean time between failures. Service density refers to the proportionality between the net output of a particular device and its gross hardware capability. Net output, in the case of a network device (e.g., switch or router), might include, for example, the number of calls handled per second. Redundancy adds to gross hardware capability but not to the net output and, thus, decreases service density. Adding hardware increases the likelihood of a failure and, thus, decreases the mean time between failures. Likewise, hot backup comes at the expense of system power. Each active element consumes some amount of the limited power available to the system. In general, the 1+1 or 1:1 redundancy designs provide the highest reliability but at a relatively high cost. Due to the importance of network availability, most network service providers prefer the 1+1 redundancy design to minimize network downtime.

In a 1:N redundancy design, instead of having one backup element per primary element, a single backup element or spare is used to backup multiple (N) primary elements. As a result, the 1:N design is generally less expensive to manufacture, offers greater service

density and better mean time between failures than the 1:1 design and requires a smaller chassis / less space than a 1:1 design. One disadvantage of such a system, however, is that once a primary element fails over to the backup element, the system is no longer redundant (i.e., no available backup element for any primary element). Another disadvantage relates to hot state backup. Because one backup element must support multiple primary elements, the typical 1:N design provides no hot state on the backup element leading to long synchronization times and, for network devices, the likelihood that connections will be dropped and availability reduced.

Even where the backup element provides some level of hot state backup it generally lacks the processing power and memory to provide a full hot state backup (i.e., 1+N) for all primary elements. To enable some level of hot state backup for each primary element, the backup element is generally a “mega spare” equipped with a more powerful processor and additional memory. This requires customers to stock more hardware than in a design with identical backup and primary elements. For instance, users typically maintain extra hardware in the case of a failure. If a primary fails over to the backup, the failed primary may be replaced with a new primary. If the primary and backup elements are identical, then users need only stock that one type of board, that is, a failed backup is also replaced with the same hardware used to replace the failed primary. If they are different, then the user must stock each type of board, thereby increasing the user’s cost.

Distributed Redundancy:

A distributed redundancy architecture spreads software backup (hot state) across multiple elements. Each element may provide software backup for one or more other elements. For software backup alone, therefore, the distributed redundancy architecture eliminates the need for hardware backup elements (i.e., spare hardware). Where hardware backup is also provided, spreading resource demands across multiple elements makes it possible to have significant (perhaps full) hot state backup without the need for a mega spare. Identical backup (spare) and primary hardware provides manufacturing advantages and customer inventory advantages. A distributed redundancy design is less expensive than many 1:1 designs and a distributed redundancy architecture also permits the location of

the hardware backup element to float, that is, if a primary element fails over to the backup element, when the failed primary element is replaced, that new hardware may serve as the hardware backup.

Software Redundancy:

In its simplest form, a distributed redundancy system provides software redundancy (i.e., backup) with or without redundant (i.e., backup) hardware, for example, with or without using backup line card 16n as discussed earlier with reference to the logical to physical card table (Fig. 11a). Referring to Fig. 29, computer system 10 includes primary line cards 16a, 16b and 16c. Computer system 10 will likely include additional primary line cards; only three are discussed herein (and shown in Fig. 29) for convenience. As described above, to load instances of software applications, the NMS creates software load records (SLR) 128a-128n in configuration database 42. The SLR includes the name of a control shim executable file and a logical identification (LID) associated with a primary line card on which the application is to be spawned. In the current example, there either are no hardware backup line cards or, if there are, the slave SRM executing on that line card does not download and execute backup applications.

As one example, NMS 60 creates SLR 128a including the executable name atm_cntrl.exe and card LID 30 (line card 16a), SLR 128b including atm_cntrl.exe and LID 31 (line card 16b) and SLR 128c including atm_cntrl.exe and LID 32 (line card 16c). The configuration database detects LID 30, 31 and 32 in SLRs 128a, 128b and 128c, respectively, and sends slave SRMs 37b, 37c and 37d (line cards 16a, 16b, and 16c) notifications including the name of the executable file (e.g., atm_cntrl.exe) to be loaded. The slave SRMs then download and execute a copy of atm_cntrl.exe 135 from memory 40 to spawn ATM controllers 136a, 136b and 136c.

Through the active query feature, the ATM controllers are sent records from group table (GT) 108' (Fig. 30) indicating how many instances of ATM each must start on their associated line cards. Group table 108' includes a primary line card LID field 447 and a backup line card LID field 449 such that, in addition to starting primary instances of

Massachusetts may be used to allow backup and primary instantiations to pass dynamic state information. If the primary instantiation fails, it can be re-started, retrieve its last known dynamic state from the backup instantiation and then initiate an audit procedure (as described above) to resynchronize with other processes. The retrieval and audit process will normally be completed very quickly, resulting in no discernable service disruption.

Although each line card in the example above is instructed by the group table to start four instantiations of ATM, this is by way of example only. The user could instruct the NMS to set up the group table to have each line card start one or more instantiations and to have each line card start a different number of instantiations.

Referring to Fig. 31a-31c, if one or more of the primary processes on element 16a (ATM 464-467) experiences a software fault (Fig. 31b), the processor on line card 16a may terminate and restart the failing process or processes. Once the process or processes are restarted (ATM 464'-467', Fig. 31c), they retrieve a copy of the last known dynamic state (i.e., backup state) from corresponding backup processes (ATM 476-479) executing on line card 16b and initiate an audit process to synchronize retrieved state with the dynamic state of associated other processes. The backup state represents the last known active or dynamic state of the process or processes prior to termination, and retrieving this state from line card 16b allows the restarted processes on line card 16a to quickly resynchronize and continue operating. The retrieval and audit process will normally be completed very quickly, and in the case of a network device, quick resynchronization may avoid losing network connections, resulting in no discernable service disruption.

If, instead of restarting a particular application, the software fault experienced by line card 16a requires the entire element to be shut down and rebooted, then all of the processes executing on line card 16a will be terminated including backup processes ATM 468-471. When the primary processes are restarted, backup state information is retrieved from backup processes executing on line card 16b as explained above. Simultaneously, the restarted backup processes on line card 16a again initiate the check-pointing

procedure with primary ATM processes 480-483 executing on line card 16c to again serve as backup processes for these primary processes. Referring to Figs. 32a-32c, the primary processes executing on one line card may be backed-up by backup processes running on one or more other line cards. In addition, each primary process may be backed-up by one or more backup processes executing on one or more of the other line cards.

Since the operating system assigns each process its own memory block, each primary process may be backed-up by a backup process running on the same line card. This would minimize the time required to retrieve backup state and resynchronize if a primary process fails and is restarted. In a computer system that includes a spare or backup line card (described below), the backup state is best saved on another line card such that in the event of a hardware fault, the backup state is not lost and can be copied from the other line card. If memory and processor limitations permit, backup processes may run simultaneously on the same line card as the primary process and on another line card such that software faults are recovered from using local backup state and hardware faults are recovered from using remote backup state.

Where limitations on processing power or memory make full hot state backup impossible or impractical, only certain hot state data will be stored as backup. The level of hot state backup is inversely proportional to the resynchronization time, that is, as the level of hot state backup increases, resynchronization time decreases. For a network device, backup state may include critical information that allows the primary process to quickly re-synchronize.

Critical information for a network device may include connection data relevant to established network connections (e.g., call set up information and virtual circuit information). For example, after primary ATM applications 464-467, executing on line card 16a, establish network connections, those applications send critical state information relevant to those connections to backup ATM applications 479-476 executing on line card 16b. Retrieving connection data allows the hardware (i.e., line card 16a) to send and

receive network data over the previously established network connections preventing these connections from being terminated / dropped.

Although ATM applications were used in the examples above, this is by way of example only. Any application (e.g., IP or MPLS), process (e.g., MCD or NS) or device driver (e.g., port driver) may have a backup process started on another line card to store backup state through a check-pointing procedure.

Hardware and Software Backup:

By adding one or more hardware backup elements (e.g., line card 16n) to the computer system, the distributed redundancy architecture provides both hardware and software backup. Software backup may be spread across all of the line cards or only some of the line cards. For example, software backup may be spread only across the primary line cards, only on one or more backup line cards or on a combination of both primary and backup line cards.

Referring to Fig. 33a, in the continuing example, line cards 16a, 16b and 16c are primary hardware elements and line card 16n is a spare or backup hardware element. In this example, software backup is spread across only the primary line cards. Alternatively, backup line card 16n may also execute backup processes to provide software backup. Backup line card 16n may execute all backup processes such that the primary elements need not execute any backup processes or line card 16n may execute only some of the backup processes. Regardless of whether backup line card 16n executes any backup processes, it is preferred that line card 16n be at least partially operational and ready to use the backup processes to quickly begin performing as if it was a failed primary line card.

There are many levels at which a backup line card may be partially operational. For example, the backup line card's hardware may be configured and device driver processes 490 loaded and ready to execute. In addition, the active state of the device drivers 492, 494, and 496 on each of the primary line cards may be stored as backup device driver

backup line card, the person acting as network manager, may prefer to have the backup line card in each of his computer systems in the same slot. In either case, where only line card 16n serves as the backup line card, once line card 16a (or any other failed primary line card) is ready to act as a primary line card again, a fail-over, as described above, is initiated from line card 16n to the primary line card to allow line card 16n to again serve as a backup line card to each of the primary line cards.

Balancing Resources:

Typically, multiple processes or applications are executed on each primary line card. Referring to Fig. 34a, in one embodiment, each primary line card 16a, 16b, 16c executes four applications. Due to physical limitations (e.g., memory space, processor power), each primary line card may not be capable of fully backing up four applications executing on another primary line card. The distributed redundancy architecture allows backup processes to be spread across multiple line cards, including any backup line cards, to more efficiently use all system resources.

For instance, primary line card 16a executes backup processes 510 and 512 corresponding to primary processes 474 and 475 executing on primary line card 16b. Primary line card 16b executes backup processes 514 and 516 corresponding to primary processes 482 and 483 executing on primary line card 16c, and primary line card 16c executes backup processes 518 and 520 corresponding to primary processes 466 and 467 executing on primary line card 16a. Backup line card 16n executes backup processes 520, 522, 524, 526, 528 and 530 corresponding to primary processes 464, 465, 472, 473, 480 and 481 executing on each of the primary line cards. Having each primary line card execute backup processes for only two primary processes executing on another primary line card reduces the primary line card resources required for backup. Since backup line card 16n is not executing primary processes, more resources are available for backup. Hence, backup line card 16n executes six backup processes corresponding to six primary processes executing on primary line cards. In addition, backup line card 16n is partially operational and is executing device driver processes 490 and storing device driver backup state 498, 500 and 502 corresponding to the device drivers on each of the primary

elements and network connection data 504, 506 and 508 corresponding to the network connections established by each of the primary line cards.

Alternatively, each primary line card could execute more or less than two backup processes. Similarly, each primary line card could execute no backup processes and backup line card 16n could execute all backup processes. Many alternatives are possible and backup processes need not be spread evenly across all primary line cards or all primary line cards and the backup line card.

Referring to Fig. 5b, if primary line card 16b experiences a failure, device drivers 490 on backup line card 16n begins using the device driver state, for example, DDS 498, corresponding to the device drivers on primary line card 16b and the network connection data, for example, CD 506, corresponding to the connections established by primary line card 16b to continue transferring network data. Simultaneously, backup line card 16n starts substitute primary processes 510' and 512' corresponding to the primary processes 474 and 475 on failed primary line card 16b. Substitute primary processes 510' and 512' retrieve active state from backup processes 510 and 512 executing on primary line card 16a. In addition, the slave SRM on backup line card 16n informs backup processes 526 and 524 corresponding to primary processes 472 and 473 on failed primary line card 16b that they are now primary processes. The new primary applications then synchronize with the rest of the system such that new network connections may be established and old network connections torn down. That is, backup line card 16n begins operating as if it were primary line card 16b.

Multiple Backup Elements:

In the examples given above, one backup line card is shown. Alternatively, multiple backup line cards may be provided in a computer system. In one embodiment, a computer system includes multiple different primary line cards. For example, some primary line cards may support the Asynchronous Transfer Mode (ATM) protocol while others support the Multi-Protocol Label Switching (MPLS) protocol, and one backup line card may be provided for the ATM primary line cards and another backup line card may

case the port cards are 1:1 or 1+1 redundant. As another example, one or more ports on port card 554a may be backed-up by separate ports on one or more port cards (e.g., port cards 554b and 554c) such that each port is 1:1 or 1+1 redundant, one or more ports on port card 554a may not be backed-up at all (i.e., not redundant) and two or more ports on 554a may be backed-up by one port on another port card (e.g., port card 554b) such that those ports are 1:N redundant. Many redundancy structures are possible using the LID to PID Card table (LPCT) 100 (Fig. 9a) and LID to PID Port table (LPPT) as described above.

Each port card includes one or more ports for connecting to external network connections. One type of network connection is an optical fiber carrying an OC-48 SONET stream, and as described above, an OC-48 SONET stream may include connections to one or more end points using one or more paths. A SONET fiber carries a time division multiplexed (TDM) byte stream of aggregated time slots (TS). A time slot has a bandwidth of 51 Mbps and is the fundamental unit of bandwidth for SONET. An STS-1 path has one time slot within the byte stream dedicated to it, while an STS-3c path (i.e., three concatenated STS-1s) has three time slots within the byte stream dedicated to it. The same or different protocols may be carried over different paths within the same TDM byte stream. In other words, ATM over SONET may be carried on an STS-1 path within a TDM byte stream that also includes IP over SONET on another STS-1 path or on an STS-3c path.

Through network management system 60 on workstation 62, after a user connects an external network connection to a port, the user may enable that port and one or more paths within that port (described below). Data received on a port card path is passed to the cross-connection card in the same quadrant as the port card, and the cross-connection card passes the path data to one of the five forwarding cards or eight port cards also within the same quadrant. The forwarding card determines whether the payload (e.g., packets, frames or cells) it is receiving includes user payload data or network control information. The forwarding card itself processes certain network control information and sends certain other network control information to the central processor over the Fast

Ethernet control bus. The forwarding card also generates network control payloads and receives network control payloads from the central processor. The forwarding card sends any user data payloads from the cross-connection card or control information from itself or the central processor as path data to the switch fabric card. The switch fabric card then passes the path data to one of the forwarding cards in any quadrant, including the forwarding card that just sent the data to the switch fabric card. That forwarding card then sends the path data to the cross-connection card within its quadrant, which passes the path data to one of the port cards within its quadrant.

Referring to Fig. 36, in one embodiment, a universal port card 554a includes one or more ports 571a-571n connected to one or more transceivers 572a-572n. The user may connect an external network connection to each port. As one example, port 571a is connected to an ingress optical fiber 576a carrying an OC-48 SONET stream and an egress optical fiber 576b carrying an OC-48 SONET stream. Port 571a passes optical data from the SONET stream on fiber 576a to transceiver 572a. Transceiver 572a converts the optical data into electrical signals that it sends to a SONET framer 574a. The SONET framer organizes the data it receives from the transceiver into SONET frames. SONET framer 574a sends data over a telecommunications bus 578a to a serializer-deserializer (SERDES) 580a that serializes the data into four serial lines with twelve STS-1 time slots each and transmits the four serial lines to cross-connect card 562a.

Each cross-connection card is a switch that provides connections between port cards and forwarding cards within its quadrant. Each cross-connection card is programmed to transfer each serial line on each port card within its quadrant to a forwarding card within its quadrant or to serial line on a port card, including the port card that transmitted the data to the cross-connection card. The programming of the cross-connect card is discussed in more detail below under Policy Based Provisioning.

Each forwarding card (e.g., forwarding card 546c) receives SONET frames over serial lines from the cross-connection card in its quadrant through a payload extractor chip

priority traffic is transmitted, and in times of traffic congestion, the traffic management chips will ensure that low priority traffic is dropped prior to any high priority traffic. The traffic management chips also perform an address translation to add the address of the traffic management chip to which the data is going to be sent by the switch fabric card. The address corresponds to internal virtual circuits set up between forwarding cards by the software and available to the traffic management chips in tables.

The traffic management chips send the modified ATM cells to switch fabric interface chips (SFIF) 589a-589n that then transfer the ATM cells to switch fabric card 570a. The switch fabric card uses the address provided by the ingress traffic management chips to pass ATM cells to the appropriate egress traffic management chips (e.g., TM 590a-590n) on the various forwarding cards. In one embodiment, the switch fabric card 570a is a 320 Gbps, non-blocking fabric. Since each forwarding card serves as both an ingress and egress, the switching fabric card provides a high degree of flexibility in directing the data between any of the forwarding cards, including the forwarding card that sent the data to the switch fabric card.

When a forwarding card (e.g., forwarding card 546c) receives ATM cells from switch fabric card 570a, the egress traffic management chips re-translate the address of each cell and pass the cells to egress bridge chips (e.g., BG 592a-592b). The bridge chips pass the cells to egress ATM interface chips (e.g., ATM IF 594a-594n), and the ATM interface chips add a re-translated address to the payload representing an ATM virtual circuit. The ATM interface chips then send the data to the payload extractor chips (e.g., payload extractor 582a-582n) that separate, where necessary, the path data into STS-1 time slots and combine twelve STS-1 time slots into four serial lines and send the serial lines back through the cross-connection card to the appropriate port card.

The port card SERDES chips receive the serial lines from the cross-connection card and de-serialize the data and send it to SONET framer chips 574a-574n. The Framers properly format the SONET overhead and send the data back through the transceivers

that change the data from electrical to optical before sending it to the appropriate port and SONET fiber.

Although the port card ports above were described as connected to a SONET fiber carrying an OC-48 stream, other SONET fibers carrying other streams (e.g., OC-12) and other types of fibers and cables, for example, Ethernet, may be used instead. The transceivers are standard parts available from many companies, including Hewlett Packard Company and Sumitomo Corporation. The SONET framer may be a Spectra chip available from PMC-Sierra, Inc. in British Columbia. A Spectra 2488 has a maximum bandwidth of 2488 Mbps and may be coupled with a 1xOC48 transceiver coupled with a port connected to a SONET optical fiber carrying an OC-48 stream also having a maximum bandwidth of 2488 Mbps. Instead, four SONET optical fibers carrying OC-12 streams each having a maximum bandwidth of 622Mbps may be connected to four 1xOC12 transceivers and coupled with one Spectra 2488. Alternatively, a Spectra 4x155 may be coupled with four OC-3 transceivers that are coupled with ports connected to four SONET fibers carrying OC-3 streams each having a maximum bandwidth of 155 Mbps. Many variables are possible.

The SERDES chip may be a Telecommunications Bus Serializer (TBS) chip from PMC-Sierra, and each cross-connection card may include a Time Switch Element (TSE) from PMC-Sierra, Inc. Similarly, the payload extractor chips may be MACH 48 chips and the ATM interface chips may be ATLAS chips both of which are available from PMC-Sierra. Several chips are available from Extreme Packet Devices (EPD), a subsidiary of PMC-Sierra, including PP3 bridge chips and Data Path Element (DPE) traffic management chips. The switch fabric interface chips may include a Switch Fabric Interface (SIF) chip also from EPD. Other switch fabric interface chips are available from Abrizio, also a subsidiary of PMC-Sierra, including a data slice chip and an enhanced port processor (EPP) chip. The switch fabric card may also include chips from Abrizio, including a cross-bar chip and a scheduler chip.

Although the port cards, cross-connection cards and forwarding cards have been shown as separate cards, this is by way of example only and they may be combined into one or more different cards.

Multiple Redundancy Schemes:

Coupling universal port cards to forwarding cards through a cross-connection card provides flexibility in data transmission by allowing data to be transmitted from any path on any port to any port on any forwarding card. In addition, decoupling the universal port cards and the forwarding cards enables redundancy schemes (e.g., 1:1, 1+1, 1:N, no redundancy) to be set up separately for the forwarding cards and universal port cards. The same redundancy scheme may be set up for both or they may be different. As described above, the LID to PID card and port tables are used to setup the various redundancy schemes for the line cards (forwarding or universal port cards) and ports. Network devices often implement industry standard redundancy schemes, such as those defined by the Automatic Protection Switching (APS) standard. In network device 540 (Fig. 35), an APS standard redundancy scheme may be implemented for the universal port cards while another redundancy scheme is implemented for the forwarding cards.

Referring again to Fig. 35, further data transmission flexibility may be provided by connecting (i.e., connections 565) each cross-connection card 562a-562b, 564a-564b, 566a-566b and 568a-568b to each of the other cross-connection cards. Through connections 565, a cross-connection card (e.g., cross-connection card 562a) may transmit data between any port or any path on any port on a universal port card (e.g., universal port cards 554a-554h) in its quadrant to a cross-connection card (e.g., cross-connection card 568a) in any other quadrant, and that cross-connection card (e.g., cross-connection card 568a) may transmit the data to any forwarding card (e.g., forwarding cards 552a-552e) or universal port card (e.g., universal port cards 560a-560h) in its quadrant. Similarly, any cross-connection card may transmit data received from any forwarding card in its quadrant to any other cross-connection card and that cross-connection card may transmit the data to any universal port card port in its quadrant.

Alternatively, the cross-connection cards in each quadrant may be coupled only with cross-connection cards in one other quadrant. For example, cross-connection cards in quadrants 1 and 2 may be connected and cross-connection cards in quadrants 3 and 4 may be connected. Similarly, the cross-connection cards in each quadrant may be coupled with cross-connection cards in only two other quadrants, or only the cross-connection cards in one quadrant (e.g., quadrant 1) may be connected to cross-connection cards in another quadrant (e.g., quadrant 2) while the cross-connection cards in the other quadrants (e.g., quadrants 3 and 4) are not connected to other cross-connection cards or are connected only to cross-connection cards in one quadrant (e.g., quadrant 2). Many variations are possible. Although these connections do not provide the flexibility of having all cross-connection cards inter-connected, these connections require less routing resources and still provide some increase in the data transmission flexibility of the network device.

The additional flexibility provided by inter-connecting one or more cross-connection cards may be used to optimize the efficiency of network device 540. For instance, a redundant forwarding card in one quadrant may be used as a backup for primary forwarding cards in other quadrants thereby reducing the number of backup modules and increasing the network device's service density. Similarly, a redundant universal port card or a redundant port on a universal port card in one quadrant may be used as a backup for primary universal port cards or ports in other quadrants. As previously mentioned, each primary forwarding card may support a different protocol (e.g., ATM, MPLS, IP, Frame Relay). Similarly, each universal port card may support a different protocol (e.g., SONET, Ethernet). A backup or spare forwarding card or universal port card must support the same protocol as the primary card or cards. If forwarding or universal port cards in one quadrant support multiple protocols and the cross-connection cards are not interconnected, then each quadrant may need multiple backup forwarding and universal port cards (i.e., one for each protocol supported). If each of the quadrants includes forwarding and universal port cards that support different protocols then each quadrant may include multiple backup forwarding and universal port cards further decreasing the network device's service density.

By inter-connecting the cross-connection cards, a forwarding card in one quadrant may serve as a backup for primary forwarding cards in its own quadrant and in other quadrants. Similarly, a universal port card or port in one quadrant may serve as a backup for a primary universal port card or port in its own quadrant and in other quadrants. For example, forwarding card 546e in quadrant 1 that supports a particular protocol (e.g., the ATM protocol) may serve as the backup forwarding card for primary forwarding cards supporting ATM in its own quadrant (e.g., forwarding cards 546a-546b) as well as for primary forwarding cards supporting ATM in quadrant 2 (e.g., forwarding cards 548b-548c) or all quadrants (e.g., forwarding card 550c in quadrant 3 and forwarding cards 552b-552d in quadrant 4). Similarly, forwarding card 548e in quadrant 2 that supports a different protocol (e.g., the MPLS protocol) may serve as the backup forwarding card for primary forwarding cards supporting MPLS in its own quadrant (e.g., forwarding cards 548a and 548d) as well as for primary forwarding cards supporting MPLS in quadrant 1 (e.g., forwarding card 546c) or all quadrants (e.g., forwarding card 550a in quadrant 3 and forwarding card 552a in quadrant 4). Even with this flexibility, to provide sufficient redundancy, multiple backup modules supporting the same protocol may be used, especially where a large number of primary modules support one protocol.

As previously discussed, each port on a universal port card may be connected to an external network connection, for example, an optical fiber transmitting data according to the SONET protocol. Each external network connection may provide multiple streams or paths and each stream or path may include data being transmitted according to a different protocol over SONET. For example, one path may include data being transmitted according to ATM over SONET while another path may include data being transmitted according to MPLS over SONET. The cross-connection cards may be programmed (as described below) to transmit protocol specific data (e.g., ATM, MPLS, IP, Frame Relay) from ports on universal port cards within their quadrants to forwarding cards within any quadrant that support the specific protocol. Because the traffic management chips on the forwarding cards provide protocol-independent addresses to be used by switch fabric

cards 570a-570b, the switch fabric cards may transmit data between any of the forwarding cards regardless of the underlying protocol.

Alternatively, the network manager may dedicate each quadrant to a specific protocol by putting forwarding cards in each quadrant according to the protocol they support. Within each quadrant then, one forwarding card may be a backup card for each of the other forwarding cards (1:N, for network device 540, 1:4). Protocol specific data received from ports or paths on ports on universal port cards within any quadrant may then be forwarded by one or more cross-connection cards to forwarding cards within the protocol specific quadrant. For instance, quadrant 1 may include forwarding cards for processing data transmissions using the ATM protocol, quadrant 2 may include forwarding cards for processing data transmissions using the IP protocol, quadrant 3 may include forwarding cards for processing data transmissions using the MPLS protocol and quadrant 4 may be used for processing data transmissions using the Frame Relay protocol. ATM data received on a port path is then transmitted by one or more cross-connection cards to a forwarding card in quadrant 1, while MPLS data received on another path on that same port or on a path in another port is transmitted by one or more cross-connection cards to a forwarding card in quadrant 3.

Policy Based Provisioning:

Unlike the switch fabric card, the cross-connection card does not examine header information in a payload to determine where to send the data. Instead, the cross-connection card is programmed to transmit payloads, for example, SONET frames, between a particular serial line on a universal port card port and a particular serial line on a forwarding card port regardless of the information in the payload. As a result, one port card serial line and one forwarding card serial line will transmit data to each other through the cross-connection card until that programmed connection is changed.

In one embodiment, connections established through a path table and service endpoint table (SET) in a configuration database are passed to path managers on port cards and service endpoint managers (SEMs) on forwarding cards, respectively. The path

managers and service endpoint managers then communicate with a cross-connect manager (CCM) on the cross-connection card in their quadrant to provide connection information. The CCM uses the connection information to generate a connection program table that is used by one or more components (e.g., a TSE chip 563) to program internal connection paths through the cross-connection card.

Typically, connections are fixed or are generated according to a predetermined map with a fixed set of rules. Unfortunately, a fixed set of rules may not provide flexibility for future network device changes or the different needs of different users / customers. Instead, within network device 540, each time a user wishes to enable / configure a path on a port on a universal port card, a Policy Provisioning Manager (PPM) 599 (Fig. 37) executing on central processor 542 selects the forwarding card port to which the port card port will be connected based on a configurable provisioning policy (PP) 603 in configuration database 42. The configurable provisioning policy may take into consideration many factors such as available system resources, balancing those resources and quality of service. Similar to other programs and files stored within the configuration database of computer system 10 described above, the provisioning policy may be modified while network device 540 is running to allow to policy to be changed according to a user's changing needs or changing network device system requirements.

When a user connects an external network connection to a particular port on a universal port card, the user notifies the NMS as to which port on which universal port card should be enabled, which path or paths should be enabled, and the number of time slots in each path. The user may also notify the NMS as to a new path and its number of time slots on an already enabled port that was not fully utilized or the user may notify the NMS of a modification to one or more paths on already enabled ports and the number of time slots required for that path or paths. With this information, the NMS fills in a Path table 600 (Figs. 37 and 38) and partially fills in a Service Endpoint Table (SET) 76' (Figs. 37 and 39).

When a record in the path table is filled in, the configuration database sends an active query notification to a path manager (e.g., path manager 597) executing on a universal port card (e.g., port card 554a) corresponding to the universal port card port LID (e.g., port 1231, Fig. 38) in the path table record (e.g., record 602).

Leaving some fields in the SET blank or assigning a particular value (e.g., zero), causes the configuration database to send an active query notification to Policy Provisioning Manager (PPM) 599. The PPM then determines – using provisioning policy 603 -- which forwarding card (FC) port or ports to assign to the new path or paths. For example, the PPM may first compare the new path's requirements, including its protocol (e.g., ATM over SONET), the number of time slots, the number of virtual circuits and virtual circuit scheduling restrictions, to the available forwarding card resources in the quadrant containing the universal port card port and path. The PPM also takes other factors into consideration including quality of service, for example, redundancy requirements or dedicated resource requirements, and balancing resource usage (i.e., load balancing) evenly within a quadrant.

As an example, a user connects SONET optical fiber 576a (Fig. 36) to port 571a on universal port card 554a and wants to enable a path with three time slots (i.e., STS-3c). The NMS assigns a path LID number (e.g., path LID 1666) and fills in a record (e.g., row 602) in Path Table 600 to include path LID 1666, a universal port card port LID (e.g., UP port LID 1231) previously assigned by the NMS and retrieved from the Logical to Physical Port Table, the first time slot (e.g., time slot 4) in the SONET stream corresponding with the path and the total number of time slots – in this example, 3 -- in the path. Other information may also be filled into Path Table 600.

The NMS also partially fills in a record (e.g., row 604) in SET 76' by filling in the quadrant number – in this example, 1 – and the assigned path LID 1666 and by assigning a service endpoint number 878. The SET table also includes other fields, for example, a forwarding card LID field 606, a forwarding card slice 608 (i.e., port) and a forwarding

card serial line 610. In one embodiment, the NMS fills in these fields with a particular value (e.g., zero), and in another embodiment, the NMS leaves these fields blank.

In either case, the particular value or a blank field causes the configuration database to send an active query notice to the PPM indicating a new path LID, quadrant number and service endpoint number. It is up to the PPM to decide which forwarding card, slice (i.e., payload extractor chip) and time slot (i.e., port) to assign to the new universal port card path. Once decided, the PPM fills in the SET Table fields. Since the user and NMS do not completely fill in the SET record, this may be referred to as a “self-completing configuration record.” Self-completing configuration records reduce the administrative workload of provisioning a network.

The SET and path table records may be automatically copied to persistent storage 21 to insure that if network device 540 is re-booted these configuration records are maintained. If the network device shuts down prior to the PPM filling in the SET record fields and having those fields saved in persistent storage, when the network device is rebooted, the SET will still include blank fields or fields with particular values which will cause the configuration database to again send an active query to the PPM.

When the forwarding card LID (e.g., 1667) corresponding, for example, to forwarding card 546c, is filled into the SET table, the configuration database sends an active query notification to an SEM (e.g., SEM 96i) executing on that forwarding card and corresponding to the assigned slice and/or time slots. The active query notifies the SEM of the newly assigned service endpoint number (e.g., SE 878) and the forwarding card slice (e.g., payload extractor 582a) and time slots (i.e., 3 time slots from one of the serial line inputs to payload extractor 582a) dedicated to the new path.

Path manager 597 and SEM 96i both send connection information to a cross-connection manager 605 executing on cross-connection card 562a – the cross-connection card within their quadrant. The CCM uses the connection information to generate a connection program table 601 and uses this table to program internal connections through one or

more components (e.g., a TSE chip 563) on the cross-connection card. Once programmed, cross-connection card 562a transmits data between new path LID 1666 on SONET fiber 576a connected to port 571a on universal port card 554a and the serial line input to payload extractor 582a on forwarding card 546c.

An active query notification is also sent to NMS database 61, and the NMS then displays the new system configuration to the user.

Alternatively, the user may choose which forwarding card to assign to the new path and notify the NMS. The NMS would then fill in the forwarding card LID in the SET, and the PPM would only determine which time slots and slice within the forwarding card to assign.

In the description above, when the PPM is notified of a new path, it compares the requirements of the new path to the available / unused forwarding card resources. If the necessary resources are not available, the PPM may signal an error. Alternatively, the PPM could move existing forwarding card resources to make the necessary forwarding card resources available for the new path. For example, if no payload extractor chip is completely available in the entire quadrant, one path requiring only one time slot is assigned to payload extractor chip 582a and a new path requires forty-eight time slots, the one path assigned to payload extractor chip 582a may be moved to another payload extractor chip, for example, payload extractor chip 582b that has at least one time slot available and the new path may be assigned all of the time slots on payload extractor chip 582a. Moving the existing path is accomplished by having the PPM modify an existing SET record. The new path is configured as described above.

Moving existing paths may result in some service disruption. To avoid this, the provisioning policy may include certain guidelines to hypothesize about future growth. For example, the policy may require small paths – for example, three or less time slots – to be assigned to payload extractor chips that already have some paths assigned instead of to completely unassigned payload extractor chips to provide a higher likelihood that

forwarding card resources will be available for large paths – for example, sixteen or more time slots -- added in the future.

Multi-Layer Network Device in One Telco Rack:

Referring again to Fig. 35, in one embodiment, each universal port card includes four ports, each of which is capable of being connected to an OC-48 SONET fiber. Since an OC-48 SONET fiber is capable of transferring data at 2.5 Giga bits per second (Gbps), each universal port card is capable of transferring data at 10 Gbps ($4 \times 2.5 = 10$). With eight port cards per quadrant, the cross-connection card must be capable of transferring data at 80 Gbps. Typically, however, the eight port cards will be 1:1 redundant and only transfer 40 Gbps. In one embodiment, each forwarding card is capable of transferring 10 Gbps, and with five forwarding cards per quadrant, the switch fabric cards must be capable of transferring data at 200 Gbps. Typically, however, the five forwarding cards will be 1:N redundant and only transfer data at 40 Gbps. With four quadrants and full redundancy (1:1 for port cards and 1:N for forwarding cards), network device 540 is capable of transferring data at 160 Gbps.

In other embodiments, each port card includes one port capable of being connected to an OC-192 SONET fiber. Since OC-192 SONET fibers are capable of transferring data at 10 Gbps, a fully redundant network device 540 is again capable of transferring 160 Gbps. In the embodiment employing one OC-192 connection per port card, each port card may include one hundred and ninety-two logical DS3 connections using sub-rate data multiplexing (SDRM). In addition, each port card may differ in its number and type of ports to provide more or less data through put. As previously mentioned, ports other than SONET ports may be provided, for example, Ethernet ports, Plesiochronous Digital Hierarchy ports (i.e., DS0, DS1, DS3, E0, E1, E3, J0, J1, J3) and Synchronous Digital Hierarchy (SDH) ports (i.e., STM1, STM4, STM16, STM64).

The universal port cards and cross-connect cards in each quadrant are in effect a physical layer switch, and the forwarding cards and switch fabric cards are effectively an upper layer switch. Prior systems have packaged these two switches into separate network

devices. One reason for this is the large number of signals that need to be routed. Taken separately, each cross-connect card 562a-562b, 564a-564b, 566a-566b and 568a-568b is essentially a switch fabric or mesh allowing switching between any path on any universal port card to any serial input line on any forwarding card in its quadrant and each switch fabric card 570a-570b allows switching between any paths on any forwarding cards. Approximately six thousand, seven hundred and twenty etches are required to support a 200 Gbps switch fabric, and about eight hundred and thirty-two etches are required to support an 80 Gbps cross-connect. Combining such high capacity multi-layer switches into one network device in a single telco rack (seven feet by nineteen inches by 24 inches) has not been thought possible by those skilled in the art of telecommunications network devices.

To fit network device 540 into a single telco rack, dual mid-planes are used. All of the functional printed circuit boards connect to at least one of the mid-planes, and the switch fabric cards and certain control cards connect to both mid-planes thereby providing connections between the two mid-planes. In addition, to efficiently utilize routing resources, instead of providing a single cross-connection card, the cross-connection functionality is separated into four cross-connection cards – one for each quadrant – (as shown in Fig. 35). Further, routing through the lower mid-plane is improved by flipping the forwarding cards and cross-connection cards in the bottom half of the front of the chassis upside down to be the mirror image of the forwarding cards and cross-connection cards in the top of the front half of the chassis.

Referring to Fig. 40, a network device 540 is packaged in a box 619 conforming to the telco standard rack of seven feet in height, nineteen inches in width and 24 inches in depth. Referring also to Figs. 41a-41c, a chassis 620 within box 619 provides support for forwarding cards 546a-546e, 548a-548e, 550a-550e and 552a-552e, universal port cards 554a-554h, 556a-556h, 558a-558h and 560a-560h, and cross-connection cards 562a-562b, 564a-564b, 566a-566b and 568a-568b. As is typical of telco network devices, the forwarding cards (FC) are located in the front portion of the chassis where network administrators may easily add and remove these cards from the box, and the universal

be connected to allow for transfer of network packets between any forwarding cards and port cards in quadrants 3 and 4.

Mid-plane 622a is also connected to external control processor cards 542b and 543b and management interface card 621. Mid-plane 622b is also connected to auxiliary processor cards 542c and 543c.

Switch fabric cards 570a and 570b are located in the back portion of chassis 620, approximately mid-way between the top and bottom of the chassis. The switch fabric cards are connected to both mid-planes 622a and 622b to allow the switch fabric cards to transfer signals between any of the forwarding cards in any quadrant. In addition, the cross-connection cards in quadrants 1 and 2 may be connected through the mid-planes and switch fabric cards to the cross-connection cards in quadrants 3 and 4 to enable network packets to be transferred between any universal port card and any forwarding card.

To provide for better routing efficiency through mid-plane 622b, forwarding cards 550a-550e and 552a-552e and cross-connection cards 566a-566b and 568a-568b in quadrants 3 and 4, located in the bottom portion of the chassis, are flipped over when plugged into mid-plane 622b. This permits the switch fabric interface 589a-589n on each of the lower forwarding cards to be oriented nearest the switch fabric cards and the cross-connection interface 582a-582n on each of the lower forwarding cards to be oriented nearest the cross-connection cards in quadrants 3 and 4. This orientation avoids having to cross switch fabric and cross-connection etches in mid-plane 622b.

Typically, airflow for cooling a network device is brought in at the bottom of the device and released at the top of the device. For example, in the back portion of chassis 620, a fan tray (FT) 626 pulls air into the device from the bottom portion of the device and a fan tray 628 blows air out of the top portion of the device. When the lower forwarding cards are flipped over, the airflow / cooling pattern is reversed. To accommodate this reversal, fan trays 630 and 632 pull air into the middle portion of the device and then fan trays 634

and 636 pull the air upwards and downwards, respectively, and blow the heated air out the top and bottom of the device, respectively.

The quadrant 3 and 4 universal port cards 558a-558h and 560a-560h may also be flipped over to orient the port card's cross-connection interface nearest the cross-connection cards and more efficiently use the routing resources. It is preferred, however, not to flip the universal port cards for serviceability reasons and airflow issues. The network managers at the telco site expect network attachments / cables to be in a certain pattern. Reversing this pattern could cause confusion in a large telco site with many different types of network devices. Also, flipping the port cards will change the airflow and cooling pattern and require a similar airflow pattern and fan tray configuration as implemented in the front of the chassis. However, with the switch fabric and internal control processor cards in the middle of the back portion of the chassis, it may be impossible to implement this fan tray configuration.

Referring to Fig. 42, mid-plane 622a includes connectors 638 mounted on the back side of the mid-plane ("back mounted") for the management interface card, connectors 640a-640d mounted on the front side of the mid-plane ("front mounted") for the quadrant 1 and 2 cross-connection cards, and front mounted connectors 642a-642b for the external control processor cards. Multiple connectors may be used for each card. Mid-plane 622a also includes back mounted connectors 644a-644p for the quadrant 1 and 2 universal port cards and front mounted connectors 646a-646j for the quadrant 1 and 2 forwarding cards.

Both mid-planes 622a and 622b include back mounted connectors 648a-648d for the switch fabric cards and back mounted connectors 650a-650d for the internal control cards. Mid-plane 622b further includes front, reverse mounted connectors 652a-652j for the quadrant 3 and 4 forwarding cards and back mounted connectors 654a-654p for the quadrant 3 and 4 universal port cards. In addition, mid-plane 622b also includes front, reverse mounted connectors 656a-656d for the quadrant 3 and 4 cross-connection cards and front mounted connectors 658a-658b for the auxiliary processor cards.

half of the chassis, alternatively, the forwarding cards and cross-connection cards in the upper half of the chassis could be flipped.

Distributed Switch Fabric:

A network device having a distributed switch fabric locates a portion of the switch fabric functionality on cards separate from the remaining / central switch fabric functionality. For example, a portion of the switch fabric may be distributed on each forwarding card. There are a number of difficulties associated with distributing a portion of the switch fabric. For instance, distributing the switch fabric makes mid-plane / back-plane routing more difficult which further increases the difficulty of fitting the network device into one telco rack, switch fabric redundancy and timing are also made more difficult, valuable forwarding card space must be allocated for switch fabric components and the cost of each forwarding card is increased. However, since the entire switch fabric need not be included in a minimally configured network device, the cost of the minimal configuration is reduced allowing network service providers to more quickly recover the initial cost of the device. As new services are requested, additional functionality, including both forwarding cards (with additional switch fabric functionality) and universal port cards may be added to the network device to handle the new requests, and the fees for the new services may be applied to the cost of the additional functionality. Consequently, the cost of the network device more closely tracks the service fees received by network providers.

Referring again to Fig. 36, as described above, each forwarding card (e.g., 546c) includes traffic management chips (e.g., 588a-588n and 590a-590b) that ensure high priority network data / traffic (e.g., voice) is transferred faster than lower priority traffic (e.g., e-mail). Each forwarding card also includes switch fabric interface (SFIF) chips (e.g., 589a-589n) that transfer network data between the traffic management chips and the switch fabric cards 570a-570b.

Referring also to Fig. 43, forwarding card 546c includes traffic management (TM) chips 588n and 590a and SFIF chips 589, and forwarding card 550a includes traffic management chips 659a and 659b and SFIF chips 660. (Fig. 43 includes only two

forwarding cards for convenience but it is to be understood that many forwarding cards may be included in a network device as shown in Fig. 35.) SFIF chips 589 and 660 on both boards include a switch fabric interface (SIF) chip 661, data slice chips 662a-662f, an enhanced port processor (EPP) chip 664 and a local timing subsystem (LTS) 665. The SFIF chips receive data from ingress TM chips 588n and 659a and forward it to the switch fabric cards 570a – 570b (Fig. 36). Similarly, the SFIF chips receive data from the switch fabric cards and forward it to the egress TM chips 590a and 659b.

Due to the size and complexity of the switch fabric, each switch fabric card 570a-570b may include multiple separate cards. In one embodiment, each switch fabric card 570a-570b includes a control card 666 and four data cards 668a-668d. A scheduler chip 670 on control card 666 works with the EPP chips on each of the forwarding cards to transfer network data between the data slice chips on the forwarding cards through cross-bar chips 672a-672l (only chips 672a-672f are shown) on data cards 668a-668d. Each of the data slice chips on each of the forwarding cards is connected to two of the cross-bar chips on the data cards. Switch fabric control card 666 and each of the switch fabric data cards 668a-668d also include a switch fabric local timing subsystem (LTS) 665, and a switch fabric central timing subsystem (CTS) 673 on control card 666 provides a start of segment (SOS) reference signal to each LTS 665 on each of the forwarding cards and switch fabric cards.

The traffic management chips perform upper level network traffic management within the network device while scheduler chip 670 on control card 666 performs the lower level data transfer between forwarding cards. The traffic management chips determine the priority of received network data and then forward the highest priority data to SIF chips 661. The traffic management chips include large buffers to store lower priority data until higher priority data has been transferred. The traffic management chips also store data in these buffers when the local EPP chip indicates that data transfers are to be stopped (i.e., back pressure). The scheduler chip works with the EPP chips to stop or hold-off data transfers when necessary, for example, when buffers on one forwarding card are close to full, the local EPP chip sends notice to each of the other EPP chips and the scheduler to

hold off sending more data. Back pressure may be applied to all forwarding cards when a new switch fabric control card is added to the network device, as described below.

The traffic management chips forward network data in predefined segments to the SIF chips. In the case of ATM data, each ATM cell is a segment. In the case of IP and MPLS, where the amount of network data in each packet may vary, the data is first arranged into appropriately sized segments before being sent to the SIF chips. This may be accomplished through segmentation and reassembly (SAR) chips (not shown).

When the SIF chip receives a segment of network data, it organizes the data into a segment consistent with that expected by the switch fabric components, including any required header information. The SIF chip may be a PMC9324-TC chip available from Extreme Packet Devices (EPD), a subsidiary of PMC-Sierra, and the data slice chips may be PM9313-HC chips and the EPP chip may be a PM9315-HC chip available from Abrizio, also a subsidiary of PMC-Sierra. In this case, the SIF chip organizes each segment of data – including header information -- in accordance with a line-card-to-switch two (LCS-2) protocol. The SIF chip then divides each data segment into twelve slices and sends two slices to each data slice chip 662a-662f. Two slices are sent because each data slice chip includes the functionality of two data slices.

When the data slice chips receive the LCS segments, the data slice chips strip off the header information, including both a destination address and quality of service (QoS) information, and send the header information to the local EPP chip. Alternatively, the SIF chip may send the header information directly to the EPP chip and send only data to the data slice chips. However, the manufacturer teaches that the SIF chip should be on the forwarding card and the EPP and data slice chips should be on a separate switch fabric card within the network device or in a separate box connected to the network device. Minimizing connections between cards is important, and where the EPP and data slice chips are not on the same card as the SIF chips, the header information is sent with the data by the SIF chip to reduce the required inter-card connections, and the data slice chips then strip off this information and send it to the EPP chip.

The EPP chips on all of the forwarding cards communicate and synchronize through cross-bar chips 674a-674b on control card 666. For each time interval (e.g., every 40 nanoseconds, "ns"), the EPP chips inform the scheduler chip as to which data segment they would like to send and the data slice chips send a segment of data previously set up by the scheduler and EPP chips. The EPP chips and the scheduler use the destination addresses to determine if there are any conflicts, for example, to determine if two or more forwarding cards are trying to send data to the same forwarding card. If a conflict is found, then the quality of service information is used to determine which forwarding card is trying to send the higher priority data. The highest priority data will likely be sent first. However, the scheduler chips include an algorithm that takes into account both the quality of service and a need to keep the switch fabric data cards 668a-668d full (maximum data through put). Where a conflict exists, the scheduler chip may inform the EPP chip to send a different, for example, lower priority, data segment from the data slice chip buffers or to send an empty data segment during the time interval.

Scheduler chip 670 informs each of the EPP chips which data segment is to be sent and received in each time interval. The EPP chips then inform their local data slice chips as to which data segments are to be sent in each interval and which data segments will be received in each interval. As previously mentioned, the forwarding cards each send and receive data. The data slice chips include small buffers to hold certain data (e.g., lower priority) while other data (e.g., higher priority) data is sent and small buffers to store received data. The data slice chips also include header information with each segment of data sent to the switch fabric cards. The header information is used by cross-bar chips 672a-672l (only cross-bar chips 672a-672f are shown) to switch the data to the correct forwarding card. The cross-bar chips may be PM9312-UC chips and the scheduler chip may be a PM9311-UC chip both of which are available from Abrizio.

Specifications for the EPD, Abrizio and PMC-Sierra chips may be found at www.pmc-sierra.com and are hereby incorporated herein by reference.

Distributed Switch Fabric Timing:

As previously mentioned, a segment of data (e.g., an ATM cell) is transferred between the data slice chips through the cross-bar chips every predetermined time interval. In one embodiment, this time interval is 40ns and is established by a 25MHz start of segment (SOS) signal. A higher frequency clock (e.g., 200 MHz, having a 5ns time interval) is used by the data slice and cross-bar chips to transfer the bits of data within each segment such that all the bits of data in a segment are transferred within one 40ns interval. More specifically, in one embodiment, each switch fabric component multiplies the 200 MHz clock signal by four to provide an 800 MHz internal clock signal allowing data to be transferred through the data slice and cross-bar components at 320 Gbps. As a result, every 40ns one segment of data (e.g., an ATM cell) is transferred. It is crucial that the EPP, scheduler, data slice and cross-bar chips transfer data according to the same / synchronized timing signals (e.g., clock and SOS), including both frequency and phase. Transferring data at different times, even slightly different times, may lead to data corruption, the wrong data being sent and/or a network device crash.

When distributed signals (e.g., reference SOS or clock signals) are used to synchronize actions across multiple components (e.g., the transmission of data through a switch fabric), any time-difference in events (e.g., clock pulse) on the distributed signals is generally termed "skew". Skew between distributed signals may result in the actions not occurring at the same time, and in the case of transmission of data through a switch fabric, skew can cause data corruption and other errors. Many variables can introduce skew into these signals. For example, components used to distribute the clock signal introduce skew, and etches on the mid-plane(s) introduce skew in proportion to the differences in their length (e.g., about 180 picoseconds per inch of etch in FR 4 printed circuit board material).

To minimize skew, one manufacturer teaches that all switch fabric components (i.e., scheduler, EPP, data slice and cross-bar chips) should be located on centralized switch fabric cards. That manufacturer also suggests distributing a central clock reference signal (e.g., 200 MHz) and a separate SOS signal (e.g., 25 MHz) to the switch fabric

etches for the reference timing signals from both the primary and redundant central timing subsystems must be matched. Compounding this with a fast clock signal and low skew component requirements makes distributing the timing very difficult.

The network device of the present invention, though difficult, includes two synchronized central timing subsystems (CTS) 673 (one is shown in Fig. 43). The etch lengths of reference timing signals from both central timing subsystems are matched to within, for example, +/- 50 mils, and both central timing subsystems distribute only reference start of segment (SOS) signals to a local timing subsystem (LTS) 665 on each forwarding card and switch fabric card. The LTSs use the SOS reference signals to generate both an SOS signal and a higher frequency clock signal. This adds components and complexity to the LTSs, however, distributing only the SOS reference signals and not both the SOS and clock reference signals significantly reduces the number of reference timing signals that must be routed across the mid-plane on matched etch lengths.

Both electro-magnetic radiation and electro-physical limitations prevent the 200 MHz reference clock signal from being widely distributed as required in a network device implementing distributed switch fabric subsystems. Such a fast reference clock increases the overall noise level generated by the network device and wide distribution may cause the network device to exceed Electro-Magnetic Interference (EMI) limitations. Clock errors are often measured as a percentage of the clock period, the smaller the clock period (5ns for a 200 MHz clock), the larger the percentage of error a small skew can cause. For example, a skew of 3ns represents a 60% error for a 5ns clock period but only a 7.5% error for a 40ns clock period. Higher frequency clock signals (e.g., 200 MHz) are susceptible to noise error and clock skew. The SOS signal has a larger clock period than the reference clock signal (40ns versus 5ns) and, thus, is less susceptible to noise error and reduces the percentage of error resulting from clock skew.

As previously mentioned, the network device may include redundant switch fabric cards 570a and 570b (Fig. 36) and as described above with reference to Fig. 43, each switch fabric card 570a and 570b may include a control card and four or more data cards.

Referring to Fig. 44, network device 540 may include switch fabric control card 666 (part of central switch fabric 570a) and redundant switch fabric control card 667 (part of redundant switch fabric 570b). Each control card 666 and 667 includes a central timing subsystem (CTS) 673. One CTS behaves as the master and the other CTS behaves as a slave and locks its output SOS signal to the master's output SOS signal. In one embodiment, upon power-up or system re-boot the CTS on the primary switch fabric control card 666 begins as the master and if a problem occurs with the CTS on the primary control card, then the CTS on redundant control card 667 takes over as master without requiring a switch over of the primary switch fabric control card.

Still referring to Fig. 44, each CTS sends a reference SOS signal to the LTSs on each forwarding card, switch fabric data cards 668a-668d and redundant switch fabric data cards 669a-669b. In addition, each CTS sends a reference SOS signal to the LTS on its own switch fabric control card and the LTS on the other switch fabric control card. As described in more detail below, each LTS then selects which reference SOS signal to use. Each CTS 673 also sends a reference SOS signal to the CTS on the other control card. The master CTS ignores the reference SOS signal from the slave CTS but the slave CTS locks its reference SOS signal to the reference SOS signal from the master, as described below. Locking the slave SOS signal to the master SOS signal synchronizes the slave signal to the master signal such that in the event that the master CTS fails and the LTSs switchover to the slave CTS reference SOS signal and the slave CTS becomes the master CTS, minimal phase change and no signal disruption is encountered between the master and slave reference SOS signals received by the LTSs.

Each of the CTS reference SOS signals sent to the LTSs and the other CTS over mid-plane etches are the same length (i.e., matched) to avoid introducing skew. The CTS may be on its own independent card or any other card in the system. Even when it is located on a switch fabric card, such as the control card, that has an LTS, the reference SOS signal is routed through the mid-plane with the same length etch as the other reference SOS signals to avoid adding skew.

Central Timing Subsystem (CTS):

Referring to Fig. 45, central timing subsystem (CTS) 673 includes a voltage controlled crystal oscillator (VCXO) 676 that generates a 25MHz reference SOS signal 678. The SOS signal must be distributed to each of the local timing subsystems (LTSs) and is, thus, sent to a first level clock driver 680 and then to second level clock drivers 682a-682d that output reference SOS signals SFC_BENCH_FB and SFC_REF1 – SFC_REFn. SFC_BENCH_FB is a local feedback signal returned to the input of the CTS. One of SFC_REF1 - SFC_REFn is sent to each LTS, the other CTS, which receives it on SFC_SYNC, and one is routed over a mid-plane and returned as a feedback signal SFC_FB to the input of the CTS that generated it. Additional levels of clock drivers may be added as the number of necessary reference SOS signals increases.

VCXO 676 may be a VF596ES50 25MHz LVPECL available from Conner-Winfield. Positive Emitter Coupled Logic (PECL) is preferred over Transistor-Transistor Logic (TTL) for its lower skew properties. In addition, though it requires two etches to transfer a single clock reference -- significantly increasing routing resources --, differential PECL is preferred over PECL for its lower skew properties and high noise immunity. The clock drivers are also differential PECL and may be one to ten (1:10) MC100 LVEP111 clock drivers available from On Semiconductor. A test header 681 may be connected to clock driver 680 to allow a test clock to be input into the system.

Hardware control logic 684 determines (as described below) whether the CTS is the master or slave, and hardware control logic 684 is connected to a multiplexor (MUX) 686 to select between a predetermined voltage input (i.e., master voltage input) 688a and a slave VCXO voltage input 688b. When the CTS is the master, hardware control logic 684 selects predetermined voltage input 688a from discrete bias circuit 690 and slave VCXO voltage input 688b is ignored. The predetermined voltage input causes VCXO 676 to generate a constant 25MHz SOS signal; that is, the VCXO operates as a simple oscillator.

Hardware control logic may be implemented in a field programmable gate array (FPGA) or a programmable logic device (PLD). MUX 686 may be a 74CBTLV3257 FET 2:1 MUX available from Texas Instruments.

When the CTS is the slave, hardware control logic 684 selects slave VCXO voltage signal 688b. This provides a variable voltage level to the VCXO that causes the output of the VCXO to track or follow the SOS reference signal from the master CTS. Referring still to Fig. 45, the CTS receives the SOS reference signal from the other CTS on SFC_SYNC. Since this is a differential PECL signal, it is first passed through a differential PECL to TTL translator 692 before being sent to MUX 697a within dual MUX 694. In addition, two feedback signals from the CTS itself are supplied as inputs to the CTS. The first feedback signal SFC_FB is an output signal (e.g., one of SFC_REF1-SFC_REFn) from the CTS itself which has been sent out to the mid-plane and routed back to the switch fabric control card. This is done so that the feedback signal used by the CTS experiences identical conditions as the reference SOS signal delivered to the LTSs and skew is minimized. The second feedback signal SFC_BENCH_FB is a local signal from the output of the CTS, for example, clock driver 682a. SFC_BENCH_FB may be used as the feedback signal in a test mode, for example, when the control card is not plugged into the network device chassis and SFC_SB is unavailable.

SFC_BENCH_FB and SFC_FB are also differential PECL signals and must be sent through translators 693 and 692, respectively, prior to being sent to MUX 697b within dual MUX 694. Hardware control logic 684 selects which inputs are used by MUX 694 by asserting signals on REF_SEL(1:0) and FB_SEL(1:0). In regular use, inputs 696a and 696b from translator 692 are selected. In test modes, grounded inputs 695a, test headers 695b or local feedback signal 698 from translator 693 may be selected. Also in regular use (and in test modes where a clock signal is not inserted through the test headers), copies of the selected input signals are provided on the test headers.

The reference output 700a and the feedback output 700b are then sent from the MUX to phase detector circuit 702. The phase detector compares the rising edge of the two input signals to determine the magnitude of any phase shift between the two. The phase

detector then generates variable voltage pulses on outputs 704a and 704b representing the magnitude of the phase shift. The phase detector outputs are used by discrete logic circuit 706 to generate a voltage on a slave VCXO voltage signal 688b representing the magnitude of the phase shift. The voltage is used to speed up or slow down (i.e., change the phase of) the VCXO's output SOS signal to allow the output SOS signal to track any phase change in the reference SOS signal from the other CTS (i.e., SFC_SYNC). The discrete logic components implement filters that determine how quickly or slowly the VCXO's output will track the change in phase detected on the reference signal. The combination of the dual MUX, phase detector, discrete logic, VCXO, clock drivers and feedback signal forms a phase locked loop (PLL) circuit allowing the slave CTS to synchronize its reference SOS signal to the master CTS reference SOS signal. MUX 686 and discrete bias circuit 690 are not found in phase locked loop circuits.

The phase detector circuit may be implemented in a programmable logic device (PLD), for example a MACH4LV-32 available from Lattice/Vantis Semiconductor. Dual MUX 694 may be implemented in the same PLD. Preferably, however, dual MUX 694 is an SN74CBTLV3253 available from Texas Instruments, which has better skew properties than the PLD. The differential PECL to TTL translators may be MC100EPT23 dual differential PECL/TTL translators available from On Semiconductor.

Since quick, large phase shifts in the reference signal are likely to be the results of failures, the discrete logic implements a filter, and for any detected phase shift, only small incremental changes over time are made to the voltage provided on slave VCXO control signal 688b. As one example, if the reference signal from the master CTS dies, the slave VCXO control signal 688b only changes phase slowly over time meaning that the VCXO will continue to provide a reference SOS signal. If the reference signal from the master CTS is suddenly returned, the slave VCXO control signal 688b again only changes phase slowly over time to cause the VCXO signal to re-synchronize with the reference signal from the master CTS. This is a significant improvement over distributing a clock signal directly to components that use the signal because, in the case of direct clock distribution, if one clock signal dies (e.g., broken wire), then the

the interrupt. Hardware control allows the CTS switch over to happen very quickly before a bad clock signal can disrupt the system.

Similarly, an activity detector 709 monitors the output of the first level clock driver 680 regardless of whether the CTS is master or slave. Instead, the output of one the second level clock drivers could be monitored, however, a failure of a different second level clock will not be detected. SFC_REF_ACTIVITY is sent from the first level clock driver to differential PECL to TTL translator 693 and then as FABRIC_REF_ACTIVITY to activity detector 709. If activity detector 709 determines that the signal is not active, which may indicate that the clock driver, oscillator or other component(s) within the CTS have failed, then it sends a signal 683b to the hardware control logic. The hardware control logic asserts KILL_CLKTREE to stop the clock drivers from sending any signals and notifies a processor chip 710 on the switch fabric control card through an interrupt. Software being executed by the processor chip detects the interrupt. The slave CTS activity detector 708 detects a dead signal from the master CTS either before or after the hardware control logic sends KILL_CLKTREE and asserts error signal 683a to cause the hardware control logic to change the input selection on MUX 686 from 688b to 688a to become the master CTS. As described below, the LTSs also detect a dead signal from the master CTS either before or after the hardware control logic sends KILL_CLKTREE and switch over to the reference SOS signal from the slave CTS either before or after the slave CTS switches over to become the master.

As previously mentioned, in the past, a separate, common clock selection signal or etch was sent to each card in the network device to indicate whether to use the master or slave clock reference signal. This approach required significant routing resources, was under software control and resulted in every load selecting the same source at any given time. Hence, if a clock signal problem was detected, components had to wait for the software to change the separate clock selection signal before beginning to use the standby clock signal and all components (i.e., loads) were always locked to the same source. This delay can cause data corruption errors, switch fabric failure and a network device crash.

predetermined threshold, then the phase detector asserts an OOL signal to the hardware control logic. The hardware control logic may immediately change the input to MUX 686 to cause the slave CTS to switch over to Master CTS and send an interrupt to the processor, or the hardware control logic may only send the interrupt and wait for software (e.g., the SRM) to determine whether the slave CTS should switch over to master.

Master / Slave CTS Control:

In order to determine which CTS is the master and which is the slave, hardware control logic 684 implements a state machine. Each hardware control logic 684 sends an IM_THE_MASTER signal to the other hardware control logic 684 which is received as a YOU_THE_MASTER signal. If the IM_THE_MASTER signal -- and, hence, the received YOU_THE_MASTER signal -- is asserted then the CTS sending the signal is the master (and selects input 688a to MUX 686, Fig. 45) and the CTS receiving the signal is the slave (and selects input 688b to MUX 686). Each IM_THE_MASTER / YOU_THE_MASTER etch is pulled down to ground on the mid-planes such that if one of the CTSs is missing, the YOU_THE_MASTER signal received by the other CTS will be a logic 0 causing the receiving CTS to become the master. This situation may arise, for example, if a redundant control card including the CTS is not inserted within the network device. In addition, each of the hardware control logics receive SLOT_ID signals from pull-down/pull-up resistors on the chassis mid-plane indicating the slot in which the switch fabric control card is inserted.

Referring to Fig. 46, on power-up or after a system or card or CTS re-boot, the hardware control logic state machine begins in INIT/RESET state 0 and does not assert IM_THE_MASTER. If the SLOT_ID signals indicate that the control card is inserted in a preferred slot (e.g., slot one), and the received YOU_THE_MASTER is not asserted (i.e., 0), then the state machine transitions to the ONLINE state 3 and the hardware control logic asserts IM_THE_MASTER indicating its master status to the other CTS and selects input 688a to MUX 686. While in the ONLINE state 3, if a failure is detected or the software tells the hardware logic to switch over, the state machine enters the OFFLINE state 1 and the hardware control logic stops asserting IM_THE_MASTER and

asserts KILL_CLKTREE. While in the OFFLINE state 1, the software may reset or reboot the control card or just the CTS and force the state machine to enter the STANDBY state 2 as the slave CTS and the hardware control logic stops asserting KILL_CLKTREE and selects input 688b to MUX 686.

While in INIT/RESET state 0, if the SLOT_ID signals indicate that the control card is inserted in a non-preferred slot, (e.g., slot 0), then the state machine will enter STANDBY state 2 as the slave CTS and the hardware control logic will not assert IM_THE_MASTER and will select input 688b to MUX 686. While in INIT/RESET state 0, even if the SLOT_ID signals indicate that the control card is inserted in the preferred slot, if YOU_THE_MASTER is asserted, indicating that the other CTS is master, then the state machine transfers to STANDBY state 2. This situation may arise after a failure and recovery of the CTS in the preferred slot (e.g., reboot, reset or new control card).

While in the STANDBY state 2, if the YOU_THE_MASTER signal becomes zero (i.e., not asserted), indicating that the master CTS is no longer master, the state machine will transition to ONLINE state 3 and the hardware control logic will assert IM_THE_MASTER and select input 688a to MUX 686 to become master. While in ONLINE state 3, if the YOU_THE_MASTER signal is asserted and SLOT_ID indicating slot 0 the state machine enters STANDBY state 2 and the hardware control logic stops asserting IM_THE_MASTER and selects input 688b to MUX 686. This is the situation where the original master CTS is back up and running. The software may reset the state machine at any time or set the state machine to a particular state at any time.

Local Timing Subsystem:

Referring to Fig. 47, each local timing subsystem (LTS) 665 receives a reference SOS signal from each CTS on SFC_REFA and SFC_REFB. Since these are differential PECL signals, each is passed through a differential PECL to TTL translator 714a or 714b, respectively. A feedback signal SFC_FB is also passed from the LTS output to both translators 714a and 714b. The reference signal outputs 716a and 716b are fed into a first

MUX 717 within dual MUX 718, and the feedback signal outputs 719a and 719b are fed into a second MUX 720 within dual MUX 718. LTS hardware control logic 712 controls selector inputs REF_SEL (1:0) and FB_SEL (1:0) to dual MUX 718. With regard to the feedback signals, the LTS hardware control logic selects the feedback signal that went through the same translator as the reference signal that is selected to minimize the effects of any skew introduced by the two translators.

A phase detector 722 receives the feedback (FB) and reference (REF) signals from the dual MUX and, as explained above, generates an output in accordance with the magnitude of any phase shift detected between the two signals. Discrete logic circuit 724 is used to filter the output of the phase detector, in a manner similar to discrete logic 706 in the CTS, and provide a signal to VCXO 726 representing a smaller change in phase than that output from the phase detector. Within the LTSs, the VCXO is a 200 MHz oscillator as opposed to the 25MHz oscillator used in the CTS. The output of the VCXO is the reference switch fabric clock. It is sent to clock driver 728, which fans the signal out to each of the local switch fabric components. For example, on the forwarding cards, the LTSs supply the 200MHz reference clock signal to the EPP and data slice chips, and on the switch fabric data cards, the LTSs supply the 200 MHz reference clock signal to the cross-bar chips. On the switch fabric control card, the LTSs supply the 200 MHz clock signal to the scheduler and cross-bar components.

The 200 MHz reference clock signal from the VCXO is also sent to a divider circuit or component 730 that divides the clock by eight to produce a 25MHz reference SOS signal 731. This signal is sent to clock driver 732, which fans the signal out to each of the same local switch fabric components that the 200 MHz reference clock signal was sent to. In addition, reference SOS signal 731 is provided as feedback signal SFC_FB to translator 714b. The combination of the dual MUX, phase detector, discrete logic, VCXO, clock drivers and feedback signal forms a phase locked loop circuit allowing the 200 MHz and 25MHz signals generated by the LTS to be synchronized to either of the reference SOS signals sent from the CTSs.

The divider component may be a SY100EL34L divider by Synergy Semiconductor Corporation.

Reference signals 716a and 716b from translator 714a are also sent to activity detectors 734a and 734b, respectively. These activity detectors perform the same function as the activity detectors in the CTSs and assert error signals `ref_a_los` or `ref_b_los` to the LTS hardware control logic if reference signal 716a or 716b, respectively, die. On power-up, reset or reboot, a state machine (Fig. 48) within the LTS hardware control logic starts in INIT/RESET state 0. Arbitrarily, reference signal 716a is the first signal considered. If activity detector 734a is not sending an error signal (i.e., `ref_a_los` is 0), indicating that that reference signal 716a is active, then the state machine changes to REF_A state 2 and sends signals over REF_SEL(1:0) to MUX 717 to select reference input 716a and sends signals over FB_SEL(1:0) to MUX 720 to select feedback input 719a. While in INIT/RESET state 0, if `ref_a_los` is asserted, indicating no signal on reference 716a, and if `ref_b_los` is not asserted, indicating there is a signal on reference 716b, then the state machine changes to REF_B state 1 and changes REF_SEL(1:0) and FB_SEL(1:0) to select reference input 716b and feedback signal 719b.

While in REF_A state 2, if activity detector 734a detects a loss of reference signal 716a and asserts `ref_a_los`, the state machine will change to REF_B state 1 and change REF_SEL(1:0) and FB_SEL(1:0) to select inputs 716b and 719b. Similarly, while in REF_B state 1, if activity detector 734b detects a loss of signal 716b and asserts `ref_b_los`, the state machine will change to REF_A state 2 and change REF_SEL(1:0) and FB_SEL(1:0) to select inputs 716a and 719a. While in either REF_A state 2 or REF_B state 1, if both `ref_a_los` and `ref_b_los` are asserted, indicating that both reference SOS signals have died, the state machine changes back to INIT/RESET state 0 and change REF_SEL(1:0) and FB_SEL(1:0) to select no inputs or test inputs 736a and 736b or ground 738. For a period of time, the LTS will continue to supply a clock and SOS signal to the switch fabric components even though it is receiving no input reference signal.

When `ref_a_los` and/or `ref_b_los` are asserted, the LTS hardware control logic notifies its local processor 740 through an interrupt. The SRM will decide, based on a failure policy, what actions to take, including whether to switch over from the master to slave CTS. Just as the phase detector in the CTS sends an out of lock signal to the CTS hardware control logic, the phase detector 722 also sends an out of lock signal OOL to the LTS hardware control logic if the magnitude of the phase difference between the reference and feedback signals exceeds a predetermined threshold. If the LTS hardware receives an asserted OOL signal, it notifies its local processor (e.g., 740) through an interrupt. The SRM will decide based on a failure policy what actions to take.

Shared LTS Hardware:

In the embodiment described above, the switch fabric data cards are four independent cards. More data cards may also be used. Alternatively, all of the cross-bar components may be located on one card. As another alternative, half of the cross-bar components may be located on two separate cards and yet attached to the same network device faceplate and share certain components. A network device faceplate is something the network manager can unlatch and pull on to remove cards from the network device. Attaching two switch fabric data cards to the same faceplate effectively makes them one board since they are added to and removed from the network device together. Since they are effectively one board, they may share certain hardware as if all components were on one physical card. In one embodiment, they may share a processor, hardware control logic and activity detectors. This means that these components will be on one of the physical cards but not on the other and signals connected to the two cards allow activity detectors on the one card to monitor the reference and feedback signals on the other card and allow the hardware control logic on the one card to select the inputs for dual MUX 718 on the other card.

Scheduler:

Another difficulty with distributing a portion of the switch fabric functionality involves the scheduler component on the switch fabric control cards. In current systems, the entire switch fabric, including all EPP chips, are always present in a network device. Registers

in the scheduler component are configured on power-up or re-boot to indicate how many EPP chips are present in the current network device, and in one embodiment, the scheduler component detects an error and switches over to the redundant switch fabric control card when one of those EPP chips is no longer active. When the EPP chips are distributed to different cards (e.g., forwarding cards) within the network device, an EPP chip may be removed from a running network device when the printed circuit board on which it is located is removed ("hot swap", "hot removal") from the network device. To prevent the scheduler chip from detecting the missing EPP chip as an error (e.g., a CRC error) and switching over to the redundant switch fabric control card, prior to the board being removed from the network device, software running on the switch fabric control card re-configures the scheduler chip to disable the scheduler chip's links to the EPP chip that is being removed.

To accomplish this, a latch 547 (Fig. 40) on the faceplate of each of the printed circuit boards on which a distributed switch fabric is located is connected to a circuit 742 (Fig. 44) also on the printed circuit board that detects when the latch is released. When the latch is released, indicating that the board is going to be removed from the network device, circuit 742 sends a signal to a circuit 743 on both switch fabric control cards indicating that the forwarding card is about to be removed. Circuit 743 sends an interrupt to the local processor (e.g., 710, Fig. 45) on the switch fabric control card. Software (e.g., slave SRM) being executed by the local processor detects the interrupt and sends a notice to software (e.g., master SRM) being executed by the processor (e.g., 24, Fig. 1) on the network device centralized processor card (e.g., 12, Fig. 1, 542 or 543, Fig. 35). The master SRM sends a notice to the slave SRMs being executed by the processors on the switch fabric data cards and forwarding cards to indicate the removal of the forwarding card. The redundant forwarding card switches over to become a replacement for the failed primary forwarding card. The master SRM also sends a notice to the slave SRM on the cross-connection card (e.g., 562-562b, 564a-564b, 566a-566b, 568a-565b, Fig. 35) to re-configure the connections between the port cards (e.g., 554a-554h, 556a-556h, 558a-558h, 560a-560h, Fig. 35) and the redundant forwarding card. The slave SRM on the switch fabric control card re-configures the registers in the scheduler

component to disable the scheduler's links to the EPP chip on the forwarding card that's being removed from the network device. As a result, when the forwarding card is removed, the scheduler will not detect an error due to a missing EPP chip.

Similarly, when a forwarding card is added to the network device, circuit 742 detects the closing of the latch and sends an interrupt to the processor. The slave SRM running on the local processor sends a notice to the Master SRM which then sends a notice to the slave SRMs being executed by the processors on the switch fabric control cards, data cards and forwarding cards indicating the presence of the new forwarding card. The slave SRM on the cross-connection cards may be re-configured, and the slave SRM on the switch fabric control card may re-configure the scheduler chip to establish links with the new EPP chip to allow data to be transferred to the newly added forwarding card.

Switch Fabric Control Card Switch-Over:

Typically, the primary and secondary scheduler components receive the same inputs, maintain the same state and generate the same outputs. The EPP chips are connected to both scheduler chips but only respond to the master / primary scheduler chip. If the primary scheduler or control card experiences a failure a switch over is initiated to allow the secondary scheduler to become the primary. When the failed switch fabric control card is re-booted, re-initialized or replaced, it and its scheduler component serve as the secondary switch fabric control card and scheduler component.

In currently available systems, a complex sequence of steps is required to "refresh" or synchronize the state of the newly added scheduler component to the primary scheduler component and for many of these steps, network data transfer through the switch fabric is temporarily stopped (i.e., back pressure). Stopping network data transfer may affect the availability of the network device. When the switch fabric is centralized and all on one board or only a few boards or in its own box, the refresh steps are quickly completed by one or only a few processors limiting the amount of time that network data is not transferred. When the switch fabric includes distributed switch fabric subsystems, the processors that are local to each of the distributed switch fabric subsystems must take part

network device. As additional forwarding cards are added to the minimal configuration - to track an increase in customer demand -- additional portions of the switch fabric are simultaneously added since a portion of the switch fabric is located on each forwarding card. Consequently, switch fabric growth tracks the growth in customer demands and fees. Also, typical network devices include 1:1 redundant switch fabric subsystems. However, as previously mentioned, the forwarding cards may be 1:N redundant and, thus, the distributed switch fabric on each forwarding card is also 1:N redundant further reducing the cost of a minimally configured network device.

External Network Data Transfer Timing:

In addition to internal switch fabric timing, a network device must also include external network data transfer timing to allow the network device to transfer network data synchronously with other network devices. Generally, multiple network devices in the same service provider site synchronize themselves to Building Integrated Timing Supply (BITS) lines provided by a network service provider. BITS lines are typically from highly accurate stratum two clock sources. In the United States, standard T1 BITS lines (2.048 MHz) are provided, and in Europe, standard E1 BITS lines (1.544 MHz) are provided. Typically, a network service provider provides two T1 lines or two E1 lines from different sources for redundancy. Alternatively, if there are no BITS lines or when network devices in different sites want to synchronously transfer data, one network device may extract a timing signal received on a port connected to the other network device and use that timing signal to synchronize its data transfers with the other network device.

Referring to Fig. 49, controller card 542b and redundant controller card 543b each include an external central timing subsystem (EX CTS) 750. Each EX CTS receives BITS lines 751 and provide BITS lines 752. In addition, each EX CTS receives a port timing signal 753 from each port card (554a-554h, 556a-556h, 558a-558h, 560a-560h, Fig. 35), and each EX CTS also receives an external timing reference signal 754 from itself and an external timing reference signal 755 from the other EX CTS.

One of the EX CTSs behaves as a master and the other EX CTS behaves as a slave. The master EX CTS may synchronize its output external reference timing signals to one of BITS lines 751 or one of the port timing signals 753, while the slave EX CTS synchronizes its output external reference timing signals to the received master external reference timing signal 755. Upon a master EX CTS failure, the slave EX CTS may automatically switch over to become the master EX CTS or software may upon an error or at any time force the slave EX CTS to switch over to become the master EX CTS.

An external reference timing signal from each EX CTS is sent to each external local timing subsystem (EX LTS) 756 on cards throughout the network device, and each EX LTS generates local external timing signals synchronized to one of the received external reference timing signals. Generally, external reference timing signals are sent only to cards including external data transfer functionality, for example, cross connection cards 562a-562b, 564a-564b, 566a-566b and 568a-568b (Fig. 35) and universal port cards 554a-554h, 556a-556h, 558a-558h, 560a-560h.

In network devices having multiple processor components, an additional central processor timing subsystem is needed to generate processor timing reference signals to allow the multiple processors to synchronize certain processes and functions. The addition of both external reference timing signals (primary and secondary) and processor timing reference signals (primary and secondary) require significant routing resources. In one embodiment of the invention, the EX CTSs embed a processor timing reference signal within each external timing reference signal to reduce the number of timing reference signals needed to be routed across the mid-plane(s). The external reference timing signals are then sent to EX LTSs on each card in the network device having a processor component, for example, cross connection cards 562a-562b, 564a-564b, 566a-566b, 568a-568b, universal port cards 554a-554h, 556a-556h, 558a-558h, 560a-560h, forwarding cards 546a-546e, 548a-548e, 550a-550e, 552a-552e, switch fabric cards 666, 667, 668a-668d, 669a-669d (Fig. 44) and both the internal controller cards 542a, 543a (Fig. 41b) and external controller cards 542b and 543b.

All of the EX LTSs extract out the embedded processor reference timing signal and send it to their local processor component. Only the cross-connection cards and port cards use the external reference timing signal to synchronize external network data transfers. As a result, the EX LTSs include extra circuitry not necessary to the function of cards not including external data transfer functionality, for example, forwarding cards, switch fabric cards and internal controller cards. The benefit of reducing the necessary routing resources, however, out weighs any disadvantage related to the excess circuitry. In addition, for the cards including external data transfer functionality, having one EX LTS that provides both local signals actually saves resources on those cards, and separate processor central timing subsystems are not necessary. Moreover, embedding the processor timing reference signal within the highly accurate, redundant external timing reference signal provides a highly accurate and redundant processor timing reference signal. Furthermore having a common EX LTS on each card allows access to the external timing signal for future modifications and having a common EX LTS, as opposed to different LTSs for each reference timing signal, results in less design time, less debug time, less risk, design re-use and simulation re-use.

Although the EX CTSs are described as being located on the external controllers 542b and 543b, similar to the switch fabric CTSs described above, the EX CTSs may be located on their own independent cards or on any other cards in the network device, for example, internal controllers 542a and 543a. In fact, one EX CTS could be located on an internal controller while the other is located on an external controller. Many variations are possible. In addition, just as the switch fabric CTSs may switch over from master to slave without affecting or requiring any other functionality on the local printed circuit board, the EX CTSs may also switch over from master to slave without affecting or requiring any other functionality on the local printed circuit board.

External Central Timing Subsystem (EX CTS):

Referring to Fig. 50, EX CTS 750 includes a T1/E1 framer/LIU 758 for receiving and terminating BITS signals 751 and for generating and sending BITS signals 752.

Although T1/E1 framer is shown in two separate boxes in Fig. 50, it is for convenience

only and may be the same circuit or component. In one embodiment, two 5431 T1/E1 Framer Line Interface Units (LIU) available from PMC-Sierra are used. The T1/E1 framer supplies 8KHz BITS_REF0 and BITS_REF1 signals and receives 8KHz BITS1_TXREF and BITS2_TXREF signals. A network administrator notifies NMS 60 (Fig. 35) as to whether the BITS signals are T1 or E1, and the NMS notifies software running on the network device. Through signals 761 from a local processor, hardware control logic 760 within the EX CTS is configured for T1 or E1 and sends an T1E1_MODE signal to the T1/E1 framer indicating T1 or E1 mode. The T1/E1 framer then forwards BITS_REF0 and BITS_REF1 to dual MUXs 762a and 762b.

Port timing signals 753 are also sent to dual MUXs 762a and 762b. The network administrator also notifies the NMS as to which timing reference signals should be used, the BITS lines or the port timing signals. The NMS again notifies software running on the network device and through signals 761, the local processor configures the hardware control logic. The hardware control logic then uses select signals 764a and 764b to select the appropriate output signals from the dual MUXs.

Activity detectors 766a and 766b provide status signals 767a and 767b to the hardware control logic indicating whether the PRI_REF signal and the SEC_REF signal are active or inactive (i.e., stuck at 1 or 0). The PRI_REF and SEC_REF signals are sent to a stratum 3 or stratum 3E timing module 768. Timing module 768 includes an internal MUX for selecting between the PRI_REF and SEC_REF signals, and the timing module receives control and status signals 769 from the hardware control logic indicating whether PRI_REF or SEC_REF should be used. If one of the activity detectors 766a or 766b indicates an inactive status to the hardware control logic, then the hardware control logic sends appropriate information over control and status signals 769 to cause the timing module to select the active one of PRI_REF or SEC_REF.

The timing module also includes an internal phase locked loop (PLL) circuit and an internal stratum 3 or 3E oscillator. The timing module synchronizes its output signal 770 to the selected input signal (PRI_REF or SEC_REF). The timing module may be an

MSTM-S3 available from Conner-Winfield or an ATIME-s or ATIME-3E available from TF systems. The hardware control logic, activity detectors and dual MUXs may be implemented in an FPGA. The timing module also includes a Free-run mode and a Hold-Over mode. When there is no input signal to synchronize to, the timing module enter a free-run mode and uses the internal oscillator to generate a clock output signal. If the signal being synchronized to is lost, then the timing module enters a hold-over mode and maintains the frequency of the last known clock output signal for a period of time.

The EX CTS 750 also receives an external timing reference signal from the other EX CTS on STRAT_SYNC 755 (one of STRAT_REF1-STRAT_REFN from the other EX CTS). STRAT_SYNC and output 770 from the timing module are sent to a MUX 772a. REF_SEL(1:0) selection signals are sent from the hardware control logic to MUX 772a to select STRAT_SYNC when the EX CTS is the slave and output 770 when the EX CTS is the master. When in a test mode, the hardware control logic may also select a test input from a test header 771a.

An activity detector 774a monitors the status of output 770 from the timing module and provides a status signal to the hardware control logic. Similarly, an activity detector 774b monitors the status of STRAT_SYNC and provides a status signal to the hardware control logic. When the EX CTS is master, if the hardware control logic receives an inactive status from activity detector 774a, then the hardware control logic automatically changes the REF_SEL signals to select STRAT_SYNC forcing the EX CTS to switch over and become the slave. When the EX CTS is slave, if the hardware control logic receives an inactive status from activity detector 774b, then the hardware control logic may automatically change the REF_SEL signals to select output 770 from the timing module forcing the EX CTS to switch over and become master.

A MUX 772b receives feedback signals from the EX CTS itself. BENCH_FB is an external timing reference signal from the EX CTS that is routed back to the MUX on the local printed circuit board. STRAT_FB 754 is an external timing reference signal from the EX CTS (one of STRAT_REF1-STRAT_REFN) that is routed onto the mid-plane(s)

and back onto the local printed circuit board such that is most closely resembles the external timing reference signals sent to the EX LTSs and the other EX CTS in order to minimize skew. The hardware control logic sends FB_SEL(1:0) signals to MUX 772b to select STRAT_FB in regular use or BENCH_FB or an input from a test header 771b in test mode.

The outputs of both MUX 772a and 772b are provided to a phase detector 776. The phase detector compares the rising edge of the two input signals to determine the magnitude of any phase shift between the two. The phase detector then generates variable voltage pulses on outputs 777a and 777b representing the magnitude of the phase shift. The phase detector outputs are used by discrete logic circuit 778 to generate a voltage on signal 779 representing the magnitude of the phase shift. The voltage is used to speed up or slow down (i.e., change the phase of) a VCXO 780 to allow the output signal 781 to track any phase change in the external timing reference signal received from the other EX CTS (i.e., STRAT_SYNC) or to allow the output signal 781 to track any phase change in the output signal 770 from the timing module. The discrete logic components implement a filter that determines how quickly or slowly the VCXO's output tracks the change in phase detected on the reference signal.

The phase detector circuit may be implemented in a programmable logic device (PLD).

The output 781 of the VCXO is sent to an External Reference Clock (ERC) circuit 782 which may also be implemented in a PLD. ERC_STRAT_SYNC is also sent to ERC 782 from the output of MUX 772a. When the EX CTS is the master, the ERC circuit generates the external timing reference signal 784 with an embedded processor timing reference signal, as described below, based on the output signal 781 and synchronous with ERC_STRAT_SYNC (corresponding to timing module output 770). When the EX CTS is the slave, the ERC generates the external timing reference signal 784 based on the output signal 781 and synchronous with ERC_STRAT_SYNC (corresponding to STRAT_SYNC 755 from the other EX CTS).

External reference signal 784 is then sent to a first level clock driver 785 and from there to second level clock drivers 786a-786d which provide external timing reference signals (STRAT_REF1-STRAT_REFN) that are distributed across the mid-plane(s) to EX LTSs on the other network device cards and the EX LTS on the same network device card, the other EX CTS and the EX CTS itself. The ERC circuit also generates BITS1_TXREF and BITS2_TXREF signals that are provided to BITS T1/E1 framer 758.

The hardware control logic also includes an activity detector 788 that receives STRAT_REF_ACTIVITY from clock driver 785. Activity detector 788 sends a status signal to the hardware control logic, and if the status indicates that STRAT_REF_ACTIVITY is inactive, then the hardware control logic asserts KILL_CLKTREE. Whenever KILL_CLKTREE is asserted, the activity detector 774b in the other EX CTS detects inactivity on STRAT_SYNC and may become the master by selecting the output of the timing module as the input to MUX 772a.

Similar to hardware control logic 684 (Fig. 45) within the switch fabric CTS, hardware control logic 760 within the EX CTS implements a state machine (similar to the state machine shown in Fig. 46) based on IM_THE_MASTER and YOU_THE_MASTER signals sent between the two EX CTSs and also on slot identification signals (not shown).

In one embodiment, ports (e.g., 571a-571n, Fig. 49) on network device 540 are connected to external optical fibers carrying signals in accordance with the synchronous optical network (SONET) protocol and the external timing reference signal is a 19.44MHz signal that may be used as the SONET transmit reference clock. This signal may also be divided down to provide an 8KHz SONET framing pulse (i.e., J0FP) or multiplied up to provide higher frequency signals. For example, four times 19.44MHz is 77.76MHz which is the base frequency for a SONET OC1 stream, two times 77.76MHz provides the base frequency for an OC3 stream and eight times 77.76MHz provides the base frequency for an OC12 stream.

The rollover counter increments on each 77.76MHz clock tick and at 9720-1 (9720-1 times 77.76MHz = 8KHz), the counter rolls over to zero. Load circuit 800 detects when the counter value is zero and loads a logic 1 into embedding registers 794a, 794b and 794c and a logic zero into embedding register 794d. As a result, the output of embedding register 794d is held high for three 77.76MHz clock pulses (since logic ones are loaded into three embedding registers) which forces the duty cycle distortion into the 19.44MHz output signal 784.

BITS circuits 802a and 802b also monitor the value of the rollover counter. While the value is less than or equal to 4860-1 (half of 8KHz), the BITS circuits provide a logic one to 8KHz output registers 798a and 798b, respectively. When the value changes to 4860, the BITS circuits toggle from a logic one to a logic zero and continue to send a logic zero to 8KHz output registers 798a and 798b, respectively, until the rollover counter rolls over. As a result, 8KHz output registers 798a and 798b provide 8KHz signals with a 50% duty cycle on BITS1_TXREF and BITS2_TXREF to the BITS T1/E1 framer.

As long as a clock signal is received over signal 781 (77.76MHz), rollover counter 796 continues to count causing BITS circuits 802a and 802b to continue toggling 8KHz registers 798a and 798b and causing load circuit 800 to continue to load logic 1110 into the embedding registers every 8KHz. As a result, the embedding registers will continue to provide a 19MHz clock signal with an embedded 8KHz signal on line 784. This is often referred to as “fly wheeling.”

Referring to Fig. 53, an extractor circuit 804 within the ERC is used to extract the embedded 8 KHz signal from ERC_STRAT_SYNC. When the EX CTS is the master, ERC_STRAT_SYNC corresponds to the output signal 770 from the timing module 768 (pure 19.44MHz), and thus, no embedded 8KHz signal is extracted. When the EX CTS is the slave, ERC_STRAT_SYNC corresponds to the external timing reference signal provided by the other EX CTS (i.e., STRAT_SYNC 755; 19.44MHz with embedded 8KHz) and the embedded 8KHz signal is extracted. The extractor circuit includes three extractor registers 806a-806c. Each extractor register is connected to the 77.76MHz

VCXO output signal 781, and on each clock pulse, extractor register 806a receives a logic one input and passes its value to extractor register 806b which passes its value to extractor register 806c which provides an 8KHz pulse 808. The extractor registers are also connected to ERC_STRAT_SYNC which provides an asynchronous reset to the extractor registers – that is, when ERC_STRAT_SYNC is logic zero, the registers are reset to zero. Every two 77.76MHz clock pulses, therefore, the extractor registers are reset and for most cycles, extractor register 806c passes a logic zero to output signal 808. However, when the EX CTS is the slave, every 8KHz ERC_STRAT_SYNC remains a logic one for three 77.76 MHz clock pulses allowing a logic one to be passed through each register and onto output signal 808 to provide an 8KHz pulse.

8KHz output signal 808 is passed to extractor circuit 804 and used to reset the rollover counter to synchronize the rollover counter to the embedded 8KHz signal within ERC_STRAT_SYNC when the EX CTS is the slave. As a result, the 8KHz embedded signal generated by both EX CTSs are synchronized.

External Local Timing Subsystem (EX LTS):

Referring to Fig. 54, EX LTS 756 receives STRAT_REF_B from one EX CTS and STRAT_REF_A from the other EX CTS. STRAT_REF_B and STRAT_REF_A correspond to one of STRAT_REF1-STRAT_REFN (Fig. 50) output from each EX CTS. STRAT_REF_B and STRAT_REF_A are provided as inputs to a MUX 810a and a hardware control logic 812 within the EX LTS selects the input to MUX 810a using REF_SEL (1:0) signals. An activity detector 814a monitors the activity of STRAT_REF_A and sends a signal to hardware control logic 812 if it detects an inactive signal (i.e., stuck at logic one or zero). Similarly, an activity detector 814b monitors the activity of STRAT_REF_B and sends a signal to hardware control logic 812 if it detects an inactive signal (i.e., stuck at logic one or zero). If the hardware control logic receives a signal from either activity detector indicating that the monitored signal is inactive, the hardware control logic automatically changes the REF_SEL (1:0) signals to cause MUX 810a to select the other input signal and send an interrupt to the local processor.

received as input 838 (marked "Alt. Input from other EX CTS"). The PLL circuit, including MUXs 772a and 772b, phase detector 776, discrete logic circuit 778 and VCXO 780, is necessary to synchronize the output of the VCXO with either output 770 of the timing module or a signal from the other EX CTS. However, PLL circuits may introduce jitter into their output signals (e.g., output 781), and passing the PLL output signal 781 via one of the STRAT_REF1-STRAT_REFN signals from one EX CTS into the PLL of the other EX CTS -- that is, PLL to PLL -- may introduce additional jitter into output signal 781. Since accurate timing signals are critical for proper data transfer with other network devices and SONET standards specifically set maximum allowable jitter transmission at interfaces (Bellcore GR-253-CORE and SONET Transport Systems Common Carrier Criteria), jitter should be minimized. Passing the output 770 of the timing module within the EX CTS to the input 838 of the other EX CTS avoids passing the output of one PLL to the input of the second PLL and thereby reduces the potential introduction of jitter.

It is still necessary to send one of the STRAT_REF1-STRAT_REFN signals to the other EX CTS (received as STRAT_SYNC 755) in order to provide ERC 782 with a 19.44MHz signal with an embedded 8KHz clock for use when the EX CTS is a slave. The ERC circuit only uses ERC_STRAT_SYNC in this instance when the EX CTS is the slave.

Layer One Test Port:

The present invention provides programmable physical layer (i.e., layer one) test ports within an upper layer network device (e.g., network device 540, Fig. 35). The test ports may be connected to external test equipment (e.g., an analyzer) to passively monitor data being received by and transmitted from the network device or to actively drive data to the network device. Importantly, data provided at a test port accurately reflects data received by or transmitted by the network device with minimal modification and no upper layer translation or processing. Moreover, data is supplied to the test ports without disrupting or slowing the service provided by the network device.

Referring to Figs. 35 and 36, network device 540 includes at least one cross-connection card 562a-562b, 564a-564b, 566a-566b, 568a-568b, at least one universal port card 554a-554h, 556a-556h, 558a-558h, 560a-560h, and at least one forwarding card 546a-546e, 548a-548e, 550a-550e, 552a-552e. Each port card includes at least one port 571a-571n for connecting to external physical network attachments 576a-576b, and each port card transfers data to a cross-connection card. The cross-connection card transfers data between port cards and forwarding cards and between port cards. In one embodiment, each forwarding card includes at least one port/payload extractor 582a-582n for receiving data from the cross-connection cards.

Referring to Fig. 56, a port 571a on a port card 554a within network device 540 may be connected to another network device (not shown) through physical external network attachments 576a and 576b. As described above, components 573 on the port card transfer data between port 571a and cross-connection card 562a, and components 563 on the cross-connection card transfer data on particular paths between the port cards and the forwarding cards or between port cards. For convenience, only one port card, forwarding card and cross-connection card are shown.

For many reasons, including error diagnosis, a service administrator may wish to monitor the data received on a particular path or paths at a particular port, for example, port 571a, and/or the data transmitted on a particular path or paths from port 571a. To accomplish this, the network administrator may connect test equipment, for example, an analyzer 840 (e.g., an Omniber analyzer available from Hewlett Packard Company), to the transmit connection of port 571b to monitor data received at port 571a and/or to the transmit connection of port 571c to monitor data transmitted from port 571a. The network administrator then notifies the NMS (e.g., NMS 60 running on PC 62, Fig. 35) as to which port or ports on which port card or port cards should be enabled and whether the transmitter and/or receiver for each port should be enabled. The network administrator also notifies the NMS as to which path or paths are to be sent to each test port, and the time slot for each path. With this information, the NMS fills in test path table 841 (Figs. 57 and 58) in configuration database 42.

corresponding to path LID 1666 and received on port 571a to port 571b and data corresponding to path LID 1666 and transmitted to port 571a by forwarding card 546c to port 571c.

Analyzer 840 may then be used to monitor both the network data received on port 571a and the network data being transmitted from port 571a. Alternatively, analyzer 840 may only be connected to one test port to monitor either the data received on port 571a or the data transmitted from port 571a. The data received on port 571a may be altered by the components on the port card(s) and the cross-connection cards before the data reaches the test port but any modification is minimal. For example, where the external network attachment 576a is a SONET optical fiber, the port card components may convert the optical signals into electrical signals that are passed to the cross-connection card and then back to the test ports, which reconvert the electrical signals into optical signals before the signals are passed to analyzer 840. Since the data received at port 571a has not been processed or translated by the upper layer processing components on the forwarding card, the data accurately reflects the data received at the port. For example, the physical layer (e.g., SONET) information and format is accurately reflected in the data received.

To passively monitor both the data received and transmitted by a particular port, two transmitters are necessary and, thus, two ports are consumed for testing and cannot be used for normal data transfer. Because the test ports are programmable through the cross-connection card, however, the test ports may be re-programmed at any time to be used for normal data transfer. In addition, redundant ports may be used as test ports to avoid consuming ports needed for normal data transfer. Current network devices often have a dedicated test port that can provide both the data received and transmitted by a working port. The dedicated test port, however, contains specialized hardware that is different from the working ports and, thus, cannot be used as a working port. Hence, although two ports may be consumed for monitoring the input and output of one working port, they are only temporarily consumed and may be re-programmed at any time. Similarly, if the port card on which a test port is located fails, the test port(s) may be quickly and easily reprogrammed to another port on another port card that has not failed.

passively monitor the data sent from and/or received by the working port, and the internal components may be capable of detecting a PRBS over the working port and/or test port(s).

Although the test ports have been shown on the same port card as the working port being tested, it should be understood, that the test ports may be on any port card in the same quadrant as the working port. Where cross-connection cards are interconnected, the test ports may be on any port card in a different quadrant so long as the cross-connection card in the different quadrant is connected to the cross-connection card in same quadrant as the working port. Similarly, the test ports may be located on different port cards with respect to each other. A different working port may be tested by re-programming the cross-connection card to multicast data corresponding to the different working port to the test port(s). In addition, multiple working ports may be tested simultaneously by re-programming the cross-connection card to multicast data from different paths on different working ports to the same test port(s) or to multiple different test ports. A network administrator may choose to dedicate certain ports as test ports prior to any testing needing to be done or the network administrator may choose certain ports as test ports when problems arise.

The programmable physical layer test port or ports allow a network administrator to test data received at or transmitted from any working port or ports and also to drive data to any upper layer card (i.e., forwarding card) within the network device. Only the port card(s) and cross-connection card need be working properly to passively monitor data received at and sent from a working port. Testing and re-programming test ports may take place during normal operation without disrupting data transfer through the network device to allow for diagnosis without network device disruption.

It will be understood that variations and modifications of the above described methods and apparatuses will be apparent to those of ordinary skill in the art and may be made without departing from the inventive concepts described herein. Accordingly, the

